VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Komprese dat s využitím transformace

Data Compression using Transformation

2018 Bc. Tomáš Řeha

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Tomáš Řeha**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Komprese dat s využitím transformace
Data Compression using Transformation

Jazyk vypracování: čeština

Zásady pro vypracování:

Cílem diplomové práce je vývoj experimentální aplikace pro bezeztrátovou kompresi dat. Komprese bude založena na Inagaki-Tomizawa-Yokoo transformaci dat.

V práci se zaměřte na:
1. Současný stav poznání v oblasti bezeztrátové komprese dat.
2. Popište výše uvedenou transformaci dat a další použité kompresní algoritmy.
3. Implementujte experimentální aplikaci.
4. Proveďte experimenty s kompresí nad vhodně zvolenými daty.
5. Získané výsledky popište.

Seznam doporučené odborné literatury:

[1] Inagaki et al., Novel and Generalized Sort-Based Transform for Lossless Data Compression, SPIRE 2009, LNCS 5721, pp. 102-113, 2009
[2] Salomon, David. Data Compression: The Complete Reference. 4th ed. London: Springer-Verlag, 2007. ISBN 1846286026.
[3] Podle pokynů vedoucího diplomové práce.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **doc. Mgr. Jiří Dvorský, Ph.D.**

Datum zadání: 01.09.2017
Datum odevzdání: 30.04.2018

doc. Ing. Jan Platoš, Ph.D.
vedoucí katedry

prof. Ing. Pavel Brandštetter, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 30. dubna 2018 .....

Na tomto místě bych rád poděkoval doc. Mgr. Jiřímu Dvorskému, Ph.D. za cenné rady a odborný dohled při zpracovávání této diplomové práce. Děkuji také Mgr. Renátě Řehové za pomoc při gramatické kontrole této práce.

Abstrakt

Hlavním cílem této diplomové práce je praktické otestování zobecněné transformace dat a její následné využití v bezeztrátové datové kompresi. Tato transformace je podrobně otestována a kombinována s rozdílnými kompresními algoritmy. Experimentální aplikace obsahuje implementované kompresní metody spolu s jednoduchým rozhraním, které umožňuje datovou kompresi pomocí argumentů příkazového řádku.

Klíčová slova: C#, Bezeztrátová komprese, Zobecněná transformace dat

Abstract

The main goal of this master's thesis is practical testing of Generalized Radix Permute data transformation and its subsequent use in loseless data compression. This transformation is thoroughly tested and combined with different compression algorithms into several variants. The experimental application includes implemented compression methods along with a simple interface that allows user to compress data using command line arguments.

Key Words: C#, Loseless compression, GRP transformation

Obsah

Seznam použitých zkratek a symbolů

BWT	–	Burrows-Wheelerova transformace
GRP	–	Generalized Radix Permute transformace
MTF	–	Move to front transformace
RLE	–	Run length encoding
SPL	–	Splay Tree komprese
HFE	–	Huffman encoding
GC	–	Garbage collector
B	–	Byte
KB	–	KiloByte
MB	–	MegaByte

Seznam obrázků

Seznam tabulek

1 Úvod

V současné době tvoří datové soubory a obecně data celkově podstatnou část informačních technologií. Uživatelé pracují s desítkami až stovkami datových souborů každý den. Informační systémy uchovávají statisíce až milióny datových záznamů. Data se taktéž člení do nespočetného množství různých druhů, formátů a velikostí. A právě velikost hraje zásadní roli u každého souboru. S rychlostí, jakou roste počet datových souborů a jejich velikost, je potřeba nějakým způsobem bojovat. Právě v této oblasti hraje zásadní roli datová komprese. Ztrátová komprese umožňuje redukovat velikost datových souborů v závislosti na jejich kvalitě. Samotné snížení kvality však nemusí být běžným uživatelem vůbec detekovatelné, ba naopak, může se jednat pouze o algoritmické vynechání nepotřebných, člověkem těžce identifikovatelných detailů. Tento způsob komprese je využíván například u obrázků, zvuku nebo videa. Bezeztrátová komprese pak vychází z úvahy, že se původní a dekomprimovaný soubor shodují. Bezeztrátová komprese má tedy za úkol daná data převést, přetransformovat, změnit jejich vnitřní strukturu tak, aby výsledná komprimovaná data byla menší než data původní. Současně také ale musíme znát způsob zpětného převodu na originální datový soubor. Bezeztrátová komprese je využívána právě tam, kde nemůžeme vynechat nepotřebné detaily. V rámci bezeztrátové komprese existuje nespočet různých metod, které se snaží rozdílnými přístupy upravovat vnitřní strukturu dat. Nejčastěji tyto metody řadíme mezi slovníkové nebo statistické kompresní algoritmy. Existují však i další algoritmy, které buď používají zcela odlišné techniky nebo kombinují výše zmíněné způsoby komprese.

Cílem této diplomové práce je nastudování a implementace experimentální zobecněné transformace dat a následné využití této transformace k plnohodnotné bezeztrátové kompresi. V rámci samotné práce je nejdříve nutné nastudovat a porozumět Generalized Radix Permute (dále jen GRP) transformaci [1], která kombinuje Burrows-Wheelerovu transformaci [2] a permutační metodu z RadixZipu [3]. Dále je potřeba transformaci naimplementovat, a následně tuto metodu otestovat z několika různých úhlů pohledu (rychlost, efektivnost, vliv parametrů na samotnou transformaci a podobně). K této transformaci je dále nutné implementovat sadu dalších kompresních metod, které přetransformovaná data upraví tak, aby skutečně docházelo k již plnohodnotné bezeztrátové kompresi (samotná transformace data nekomprimuje). Naimplementované kompresní metody je následně nutné porovnat, vyhodnotit a na základě výsledků pak zvolit nejlepší možnou kombinaci metod pro vytvoření finální verze experimentální aplikace.

Samotným výsledkem této práce je knihovna s naimplementovanou experimentální transformační metodou GRP a patřičné algoritmy pro bezeztrátovou kompresi. Textová část obsahuje stručný popis datové komprese. V textové části je dále obsažen detailní popis zvolených algoritmů, samotné transformace, použitých komprimačních metod a detailů testování včetně interpretace výsledků. Knihovna obsahuje funkce pro samotnou transformaci dat, kterým předáváme vstupní řetězec nebo pole Bytu. Výstupem funkcí jsou transformovaná data ve stejném formátu. Knihovna taktéž obsahuje další funkce pro celkovou kompresi s využitím zmíněné transformace.

2 Současný stav bezeztrátové komprese

V této kapitole bude popsán současný stav poznání v oblasti bezeztrátové komprese dat. V první části budou stručně objasněny různé přístupy bezeztrátové datové komprese. V druhé části budou popsány jednotlivé typy algoritmů využívané při bezeztrátové kompresi. Veškeré informace uvedeny v této kapitole byly čerpány z odborné literatury:

- Fundamental Data Compression – Ida Pu [4]
- Introduction to data compression – Khalid Sayood [5]
- Data Compression, The Complete Refference - David Salomon [6]

2.1 Komprese dat

Datová komprese představuje vědu, která úzce souvisí s informačními technologiemi. A právě datová komprese hraje nesmírně důležitou roli v technologickém pokroku. Bez datové komprese by samotný technologický pokrok Internetu, digitálních technologií, telekomunikačních věd a dalších odvětví byl značně zpomalen. Datová komprese se tedy využívá téměř všude. Mezi možné příklady pak můžeme zařadit kompresi dokumentů, obrázků, videa, zvuku, signálů a dalších datových souborů. Konkrétních příkladů však existuje nespočet.

Cílem datové komprese je redukce samotné velikosti vstupních dat za účelem celkově snížené velikosti dat výstupních. Současně ale musíme znát způsob, jak zkomprimovaná data zpětně převést na původní vstupní data. Datová komprese hledá nejčastěji v datech určitou podobnost, tedy vzory, které se v datech redundantně vyskytují. V rámci komplexnějších metod jsou tyto vzory deterministickými pravidly rovnou vytvářeny. Pro představu uvažujme příklad, kdy máme textový soubor a v něm se vyskytuje jedna dlouhá fráze, přičemž v celém textu se daná fráze vyskytuje hned několikrát. Tuto frázi můžeme zapsat do tabulky a dále do tabulky zaznačíme pozice, na kterých se daná fráze v původních textových datech vyskytovala. Následně z původního textu odstraníme všechny výskyty naší dlouhé fráze. Ačkoli tabulka s pozicemi zabírá nějaký datový prostor, je určitě jasné, že celková velikost upraveného textového souboru a naší vytvořené tabulky bude redukována. Samozřejmě tento způsob půjde využít v textových dokumentech, pokud však uvážíme například obrázek, tak bychom se stejným způsobem moc daleko nedostali. V rámci komprese obrazu musíme využívat předvídatelnosti. Pokud jsme schopni nějakým způsobem izolovat určité části obrázku s podobnými vlastnostmi, jsme pak schopni tento obrázek lépe komprimovat. Pokud si představíme fotografii, kde polovinu celé plochy zabírá modrá obloha, pak jsme určitě schopni tuto oblast daleko lépe komprimovat, než pokud uvažujeme zcela náhodný obrázek s velkým množstvím barevných přechodů. Samotná komprese je reprezentována tak, že jsou specifikovány určité vlastnosti o celé oblasti, a ne o jednotlivých pixelech. Tyto metody jsou samozřejmě velice zjednodušené a poskytují čtenáři spíše náhled na

danou problematiku datové komprese, než aby reprezentovali konkrétní komprimační metody. Ty jsou obvykle daleko komplikovanější.

Jak již bylo řečeno, datová komprese je nesmírně důležitá, protože napomáhá k efektivnímu využití datového uložiště. S tím je obvykle spojeno i efektivní využití prostředků, které jsou využívány k přenosu dat (například přenos přes síť). Tím se snižuje nejenom množství přenesených dat, ale i čas potřebný k samotnému přenosu. S pomocí datové komprese jsme tedy schopni uložit větší počet dat a snížit tak náklady. Na druhou stranu je nutné poznamenat, že komprimační a dekomprimační proces vyžaduje určité množství výpočetních prostředků. Pokud na jedné straně sítě data zkomprimujeme, musíme je na druhé straně i dekomprimovat. To může být v určitých případech (zvláště při zvolení špatné komprimační metody) nežádoucí a celý proces přenosu dat může být naopak pomalejší. Z tohoto důvodu je potřeba nalézt algoritmus s ideálním kompresním poměrem, dobou samotné komprimace i dekomprimace a množstvím požadovaných výpočetních prostředků.

Výkonnost jednotlivých kompresních algoritmů můžeme měřit několika různými způsoby. Nejpoužívanější způsob představuje obecné porovnávání kompresního poměru (tedy rozdíl mezi velikostí výstupních a vstupních dat). Obecně je ale porovnávání jednotlivých kompresních algoritmů poměrně složité, protože kompresní algoritmy jsou ve velkém množství závislé na vstupních datech. Proto jsou obvykle k testování používány kompresní korpusy. Ty obsahují hned několik rozdílných souborů. Výsledek nad konkrétním kompresním korpusem je pak daleko přesvědčivější a celkově lépe shrnuje výkonnost algoritmu nad rozdílnými daty. Z dalších vlastností je vždy vhodné sledovat i čas samotného výpočtu. Mezi další užívaná kritéria můžeme zařadit údaje o kompresi, dekompresi, využité paměti, komplexitě algoritmu a kvalitě (pokud používáme ztrátovou kompresi). V některých případech může být dále vhodné sledovat entropii, overhead a celkovou efektivnost kompresních metod. Pro základní metody porovnávání výkonnosti kompresních algoritmů musíme nejdříve definovat:

- Velikost vstupních dat, kterou označujeme jako i.
- Velikost výstupních dat, které označujeme jako o.
- Kompresní poměr, který představuje jednoduchý poměr mezi velikostí výstupních a vstupních dat. Kompresní poměr je dále označován jako α (1).
- Kompresní faktor, který představuje opačnou hodnotu kompresního poměru a je dále označován jako f (2).
- Procentuální úsporu, tedy hodnotu představující procentuálně ušetřenou velikost. Označujeme ji dále jako s (3).
- Aritmetický průměr, který dále značíme pomocí $\overline{x}$.
- Shannonova entropie, kterou značíme dále jako H (4).

Pro výše definované hodnoty pak platí:

$$\alpha = \frac{o}{i} \tag{1}$$

$$f = \frac{i}{o} \tag{2}$$

$$s = \frac{i - o}{i} \tag{3}$$

$$H = -\sum P(s_i) \log_2 P(s_i) \tag{4}$$

Pro příklad jednoduchého porovnání uvažujme obrázek s rozlišením 256x256 pixelů. Vstupní data mají velikost 65 536 Bytu. Výstupní (komprimovaná) data pak 16 384 Bytu. Na základě výše definovaných vzorců je kompresní poměr roven 0,25. Kompresní faktor je 4 a procentuální úspora představuje 75

Celá datová komprese se dělí na dvě hlavní části, ztrátovou a bezeztrátovou datovou kompresi. Každá z těchto částí je pak obvykle používána na rozdílná vstupní data. Jak již název napovídá, hlavním rozdílem je schopnost úplné rekonstrukce původních vstupních dat, ale o tom bude zmínka v dalších podkapitolách.

2.2 Ztrátová komprese dat

Ztrátová datová komprese je taková datová komprese, u které není možné z komprimovaných dat zpětně rekonstruovat původní vstupní data. U ztrátové komprese dat nejsme při použití dekompresní části schopni rekonstruovat nepodstatné a zanedbané detaily, které byly v rámci komprimační metody ztraceny. Pomocí ztrátové komprese jsou pak nejčastěji komprimovány obrázky, zvuk nebo video. Jednoduchým příkladem správného využití ztrátové komprese může být například zvukový soubor obsahující několik zvukových stop. Jedna zvuková stopa je pak mimo frekvenční rozsah, který je člověk schopný detekovat. Taková zvuková stopa může být v rámci ztrátové komprese úplně vypuštěna, čímž se sníží i celková velikost celého zvukového souboru. Zpětná rekonstrukce pak ale bude nemožná, právě díky úplnému vypuštění již zmiňované zvukové stopy.

Aproximovaný rekonstrukční proces má samozřejmě své výhody i nevýhody. Jednou z největších výhod je opravdu znatelné zlepšení kompresního poměru. Na druhou stranu tímto způsobem můžeme značně ovlivnit samotnou kvalitu dat. Pokud je v dané zvukové stopě odstraněna pouze neslyšitelná část, nebo pokud je v obrázku odstraněn odstín pouze několika málo pixelu, jsou tyto změny dostatečně malé a poněkud zanedbatelné. V případě, kdy kompresní metoda začne zanedbávat i důležité detaily, je ovlivněna i kvalita samotných dat, což je samozřejmě velice nežádoucí. U ztrátové komprese je proto velice důležitá i volba výsledné kvality zkomprimovaných dat. Jednou z přívětivých vlastností je však poměrně dobrý poměr ušetřené velikosti

k celkové kvalitě výsledných dat. Pěkným příkladem je například obrázkový formát JPEG, kde 75 % kvalita původních nekomprimovaných dat ušetří přibližně polovinu celkové velikosti finálního souboru, přičemž samotná kvalita obrázku je pořád označována jako dostačující [7].

Samotné určování kvality je však subjektivní. Ideálním případem by bylo porovnání výsledků několika kompresních metod s různou kvalitou jedním člověkem, který subjektivně vybere nejlepší výsledek komprese. To samozřejmě ale není možné. Většina dnešních softwarových a technologických korporací používá fixní nastavení pro téměř všechny používané kompresní metody, přičemž u všech se nastavení kvality mírně liší. Pro představu jsou pro formát JPEG uvedeny hodnoty používané kvality 1.

Tabulka 1: Používaná kvalita JPEG komprese technologickými společnostmi

Korporace/společnost	Použití	Kvalita v %
Google	Náhledy	74-76
Facebook	Fotografie v plném rozlišení	69-91
Yahoo	Obrázky na úvodních stránkách	74-76
Youtube	Obrázky na úvodních stránkách	70-82
Wikipedia	Všechny obrázky	80
Microsoft (Windows)	Pozadí plochy	82

Jak je vidět z tabulky, tak se ani známé společnosti jako Facebook a Google neshodují na ideální procentuální kvalitě obrázku. Fixní přístup, kdy je procentuální kvalita nastavena na určitou konstantu, představuje poměrně bezpečné a stabilní řešení. Nicméně, pokud je zvolen fixní přístup, je vizuální kvalita nejednotná. V rámci fixního nastavení budou vždy existovat soubory, pro které je nastavení kvality příliš nízké, což má za následek nekvalitní a nevěrohodné vizuální podání. Stejně tak ale budou existovat soubory, pro které bude stejné nastavení zbytečně vysoké, čímž prakticky zbytečně plýtváme úložným prostorem. Existují tak i algoritmy, které se snaží tento problém adaptivně řešit. Tyto algoritmy obvykle pracují v zadaném rozsahu kvality, přičemž se snaží nalézt nejoptimálnější parametry pro cílovou kompresní metodu. Adaptivní algoritmy pro detekování vhodné výsledné kvality pak mohou volit lepší, ale i horší parametry a celkové nastavení kompresních algoritmů pro konkrétní vstupní data tak reprezentuje nestabilní řešení. [7]

2.3 Bezeztrátová komprese dat

Bezeztrátová datová komprese představuje druhý základní přístup v kompresi dat. Za bezeztrátovou kompresi můžeme považovat pouze takové kompresní metody, které garantují úplnou a přesnou rekonstrukci originálních dat z komprimované verze. Při samotné kompresi tedy neredukujeme data o nepotřebné detaily, jako tomu bylo v případě ztrátové komprese, ale snažíme se nějakým způsobem zbavit redundantních informací, nebo tyto redundantní části dat zapsat tak, aby zabíraly co možná nejmenší možnou velikost.

Bezeztrátová komprese je vhodná pro všechny typy dat, kde i sebemenší vypuštění jakékoliv informace znehodnocuje všechna data. Obvykle se používá ke kompresi textových dokumentů a všeobecně všech textově založených dat. Tím jsou myšlena jakákoliv data, která využívají nějakou formu textového zápisu. V ostatních případech se může jednat o binární soubory, databáze, spustitelné soubory a další. Pomocí bezeztrátové komprese mohou být například komprimovány i zvukové stopy, obrázky nebo video. U tohoto typu dat je pak bezeztrátová komprese využívána zvláště v případě, kdy je potřeba zachovat originální kvalitu dat. Může se tak jednat například o důkazní materiál nebo medicínské snímky. Nejvíce výhodná je pak bezeztrátová komprese textových dat s omezeným počtem možných znaků. V takovém případě lze zcela efektivně využít jeden Byte pro zápis více než jednoho textového znaku, čímž se výsledná velikost několikanásobně snižuje. Za vhodný příklad můžeme považovat kompresi biomedicínských sekvencí DNA a RNA. Konkrétních příkladů by se ale našla celá řada.

Bezeztrátová komprese obsahuje celou řadu kompresních algoritmů a metod, které se podle způsobu komprese dělí na jednotlivé kategorie. Ve výsledné kompresi je obvykle využito hned několik různých algoritmů, které data postupně zpracovávají. Každý takový algoritmus však může patřit do jiné kategorie. Mezi kategorie bezeztrátové komprese můžeme zařadit transformace, slovníkové algoritmy, statistické algoritmy a ostatní algoritmy, které nespadají do žádné z vyjmenovaných kategorií. Každá z těchto kategorií obsahuje hned několik typických algoritmů, které takovou kategorii reprezentují.

2.3.1 Transformační metody

Transformační metody představují typ algoritmů, který pouze napomáhá k uspořádání dat takovým způsobem, jenž zefektivňuje ostatní kompresní metody. Transformační algoritmy tedy téměř nikdy neprovádí samotnou kompresi. Pokud je komprese prováděna transformační metodou, jedná se spíše o hybridní metodu, která do této kategorie patří jen z části. Typický transformační algoritmus obvykle vstupní data přepisuje do určitého typu datové struktury. Nejčastěji se jedná o pole, matici nebo graf (případně se může jednat i o jednotlivé varianty grafu, jako je například strom). Využity však mohou být i další typy datových struktur. Takto rozložená data v konkrétní struktuře jsou podle konkrétních pravidel dané transformace dále upravována. Velké množství transformačních algoritmů využívá blokových transformací. Bloková transformace představuje reorganizaci dat v použité struktuře na základě definovaných pravidel. V konkrétním příkladu může být blok reprezentován části pole, která je definovaná na základě indexů v poli. Za blok můžeme považovat i sloupcový nebo řádkový vektor (sloupec, řádek) v matici. V kontextu teorie grafů to může být třeba konkrétní podgraf. Tyto blokové části jsou poté například na základě setřídění klíčů různě upravovány, což žádoucím způsobem pozměňuje data. Data výsledné transformace pak nejčastěji obsahují určité vlastnosti, které poté využívají jiné kompresní algoritmy. Téměř všechny transformace jsou proto používány v kombinaci s jiným kompresním algoritmem. Zvolený kompresní algoritmus, který data dále upravuje po provedené transformaci, je volen na základě konkrétních vlastností. Pro jednoduchý příklad

uvažujme o transformaci, která pomocí blokových úprav transformuje vstupní data tak, aby data výstupní obsahovala sekvence stejných znaků blízko u sebe. Kompresní algoritmus může využít této vlastnosti a tyto sekvence případně nahradit kratší sadou znaků.

Ke každé transformaci musí být definovaná i transformace inverzní, která transformuje výstupní data dopředné transformace zpět na data původní. Inverzní transformace je stejně důležitá jako transformace dopředná. Je vhodné poznamenat, že obě tyto transformace mohou obsahovat naprosto rozdílný postup výpočtu. Z tohoto důvodu mohou být i obě transformace rozdílně náročné na samotný výpočet. Některé kompresní nástroje jsou právě na základě této skutečnosti konstruovány. Tedy na základě takové skutečnosti, kdy je jedna z transformací několikanásobně náročnější z hlediska časového výpočtu než transformace druhá.

Za snad nejrozšířenější algoritmus z této kategorie můžeme označit Burrows-Wheelerovu transformaci [2] (dále jen BWT). BWT používá (jako obecně většina transformačních algoritmů) metody blokového třídění. Na konec vstupních dat se nejprve umístí zarážka (symbol označující konec dat). Pro vstupní data jsou následně vytvořeny cyklické rotace všech vstupních dat, ty mohou být pro názornost umístěny například do matice. Následně jsou lexikograficky (po řádcích) seřazeny cyklické rotace a na závěr je zapsán poslední sloupec (poslední sloupcový vektor) ve zmíněné matici. Tím je dopředná transformace vykonána. Do inverzní transformace pak vstupuje výsledný řetězec dopředné transformace. Ten je umístěn jako poslední sloupcový vektor do nové matice. Poté se lexikograficky seřadí celá matice na základě vloženého sloupcového vektoru. Tento proces se opakuje n-krát, kde n je délka vstupního řetězce. Po dokončení procesu získáváme původní řetězec. Cílem této diplomové práce je implementace GRP transformace, která má hned několik společných vlastností s výše popsanou BWT jako je například přepis do matice, lexikografické blokové třídění a podobně. Celý proces GRP je pak rozepsán v kapitole číslo 3.1.

2.3.2 Statistické kompresní algoritmy

V předchozí části byly rozebrány transformační algoritmy, které slouží výhradně jako pomocné metody a obvykle data nekomprimují. Statistické kompresní algoritmy, na rozdíl od transformací, však už data komprimují. Jak již název napovídá, statistické kompresní algoritmy vycházejí z pravděpodobnosti výskytu jednotlivých znaků. V přirozených datech jsou zcela jistě obsaženy některé znaky častěji než jiné. Uvažujme jednoduchý česky psaný textový dokument. V průměru je pro každý česky psaný text nejvíce frekventované písmeno e (téměř každý desátý znak) [8]. Na druhou stranu nejméně frekventovanými znaky jsou w nebo q [8]. Pokud této skutečnosti využijeme a nahradíme nejfrekventovanější znak nejkratší možnou posloupností bitu, pak zcela určitě provádíme kompresi dat. Stejný princip jde aplikovat i na jiná než textová data a konkrétní znaky. V případě obrázků můžeme používat místo konkrétních znaků pixely. Stejně tak nemusí být tento princip omezen na konkrétní atomické jednotky (pixely, znaky). V propracovanějších metodách lze stejnou metodu aplikovat na dvojznaky. V kontextu obrázkové komprese to může být posloupnost pixelů nebo pixel a jeho okolí, tedy pixely sousedící.

Použití statistických kompresních metod však vede k záměně cílového kódování. Jednotlivé znaky nebo pixely již nejsou reprezentovány stejně dlouhou posloupnosti bitů. Pro aplikaci této metody musíme zavést zápis s proměnlivým počtem bitů. Pokud byl v původním textu každý znak reprezentován jedním Bytem, jsou po následné kompresi výsledné znaky zapisovány rozdílně dlouhou posloupnosti bitů. Z tohoto důvodu musí být při použití tohoto typu metod vždy přesně definováno, jak jednoznačně a spolehlivě dekódovat takto zapsaná data. Současně je ale také žádoucí, aby průměrná délka zápisu jednotlivých znaků nebo pixelů byla co nejkratší. Tento způsob komprese je taktéž nazýván jako entropické kódování a je využíván převážně pro bezeztrátovou kompresi. Ve výsledku existuje celá řada kompresních metod využívající statistických předpokladů o datech. Mezi ty nejznámější patří Shannon-Fanovo kódování, Huffmanovo kódování, Aritmetické kódování, Fibbonaciho kódování a další. Huffmanovo kódování je dále rozebráno v kapitole číslo 3.5 a je dále použito jako jedna z variant ve výsledné kompresní aplikaci.

2.3.3 Slovníkové kompresní algoritmy

V předchozí části jsme definovali principy statistických kompresních algoritmů. Ty využívaly statistického a pravděpodobnostního předpokladu o určitých atomických částech (znaky, pixely) a na základě této skutečnosti definovaly zápis pomocí rozdílně dlouhé posloupnosti bitů. Slovníkové kompresní algoritmy na rozdíl od těch statistických využívají fixní kódování (znaky a pixely jsou vždy zapisovány pomocí stejně dlouhé posloupnosti bitů) a kompresní proces provádějí za pomoci slovníků. Ve slovníku jsou umístěny vzory, které se běžně vyskytují ve vstupních datech. Při kompresi jsou vstupní data iteračně procházena. Následně nalezené vzory jsou nahrazeny patřičným indexem ze slovníku. Tímto způsobem pak vzniká menší komprimovaný formát dat. Efektivnost samotné komprese závisí na redundanci ve vstupních datech. Důležité je i definování správných vzorů umístěných ve slovníku. Čím více se konkrétní vzor vyskytuje ve vstupních datech, tím více jsme schopni daná data komprimovat. Slovník může být definován buď staticky, semi-adaptivně nebo adaptivně.

Výběr staticky definovaného slovníků je vhodný, pokud máme určitou znalost o vstupních datech. Pokud tedy dopředu víme, jaká data budeme komprimovat, můžeme tomu přizpůsobit i slovník. Ten pak může být mnohdy nejlepším řešením právě pro specifické případy. Pro příklad si můžeme představit kompresi anglicky psaného textu. Slovník bude obsahovat půl milionů anglicky psaných slov. V případě, kdy narazíme na slovo, u kterého najdeme shodu se záznamem ze slovníků, můžeme pak v původních datech toto slovo nahradit indexem, který odkazuje do slovníků. Všeobecně však může existovat nespočet různých situací, kdy bude staticky definovaný slovník nejvíce efektivním řešením. Bohužel toto řešení je vhodné pouze pro taková data, pro která byl daný slovník vytvořen. Pokud bychom pomocí stejného slovníku komprimovali jiná data, byl by výsledek pravděpodobně nulový. Semi-adaptivní slovník je vytvářen při prvním průchodu daty, v něm jsou spočítány konkrétní výskyty jednotlivých slov a na základě seřazených

frekvencí je pak sestaven slovník. V druhém průchodu se následná slova ve vstupních datech nahrazují patřičnými indexy ve slovníku.

Adaptivně definovaný slovník má základ ve dvou velice známých kompresních algoritmech LZ77 [9] a LZ78 [10]. Algoritmy popisují rozdílný přístup při vytváření adaptivního slovníku za průběhu komprese. Na základě popularity těchto algoritmů existuje i celá řada jejich variant. Kompresní algoritmus LZ77 prochází vstupní data pouze jednou. Kompresní metoda využívá tzv. posuvné okénko, které je rozděleno na dvě části. Jednou částí je vyhledávací okno (search buffer) a částí druhou je pohyblivé (look-ahead buffer) okno. Jejich velikost je konstantní a vzhledem ke vstupní datům malá. Do pohyblivého okna se nejdříve načte začátek vstupu. Posléze je v každém cyklu vyhledáváno nejdelší slovo ve vyhledávacím okně, které je shodné s prefixem v pohyblivém okně. Po nalezení takového slova je následně slovo zakódováno do trojice $<i, j, z>$ kde i reprezentuje vzdálenost slova od začátku pohyblivého okna, j reprezentuje délku shody a z první neshodný znak. Pokud bychom tuto trojici kódovali pomocí 3 Bytu (12 bitu pro znakový rozestup, 4 bity pro délku shody a 8 bitů pro neshodný znak), pak jsme schopni se odkazovat na druhý výskyt slova až do vzdálenosti 4096 předchozích znaků. Délka shody může mít až 15 znaků, což obecně dostatečně pokrývá obvyklé shody jednotlivých řetězců. V případě delší shody může být daný řetězec bez problémů reprezentován více než jednou zakódovanou dvojící. Samotné porovnávání se provádí pomocí pohyblivého a vyhledávacího okna.

Algoritmus LZ78 je nástupcem předchozího LZ77. Místo posuvného okénka je tento algoritmus založen na principu ukládání frází vstupního textu do vytvořeného slovníku. V případě, kdy je fráze ve vstupním textu obsažena znovu, je pak fráze nahrazena výstupní dvojící $<i, a>$, kde i reprezentuje index ve slovníku a symbol a označuje znak, který následuje za nalezenou frází. LZ78 je paměťově velice náročná metoda, existuje však několik způsobů, jak tento problém řešit. Pokud slovník zaplní přidělenou paměť, je obvykle použita metoda zmrazení nebo smazání. V prvním případě je slovník uzavřen a už do něj nejsou přidávány žádné nové záznamy, přičemž dále probíhá komprese s již vytvořeným a uzavřeným slovníkem. V druhém případě se slovník smaže. Kompresní metoda zde však nekončí, nýbrž pokračuje zcela stejným způsobem jako doposud. Slovník je tedy znovu sestavován a v případě dalšího naplnění opět smazán. Existují však i další způsoby, jak tyto problémy řešit. Nejznámější modifikací LZ78 je algoritmus LZW [11]. Ten byl z počátku patentován a je například použit v obrázkovém formátu GIF.

3 Popis použitých algoritmů

V této kapitole si podrobně představíme a rozebereme všechny použité algoritmy, které jsou součástí naší kompresní experimentální aplikace. V první části bude představena GRP transformace včetně podrobného popisu a jednoduché ukázky výpočtu na konkrétním vstupním řetězci. V rámci této kapitoly budou dále představeny algoritmy pro samotnou kompresi dat.

3.1 Generalized radix permute transformace

Kompresní algoritmy využívající blokového třídění byly v historii analyzovány a vyhodnocovány jak teoreticky, tak i empiricky výzkumníky z oblastí teoretické informatiky a algoritmických výpočtů [2]. Tyto algoritmy byly v navazujících publikacích dále rozšiřovány a modifikovány [2]. Nejvíce těchto rozšíření je založeno na modifikaci BWT, která představuje základ pro kompresní algoritmy využívající blokového třídění [12]. Současně bylo v nedávné publikaci [3] ukázáno, že příklady v rámci RadixZip transformace mohou plně nahradit kompresní algoritmy využívající blokové třídění, a tedy i konkrétní BWT transformaci [3]. Z této úvahy pak vychází Generalized radix permute (GRP) transformace. Ta představuje kombinaci BWT a permutační transformace používané v RadixZipu. GRP tedy přejímá vlastnosti obou těchto transformací a ty kombinuje v unikátní parametrickou transformační metodu [3].

GRP transformace tvoří úplný základ naší knihovny pro bezeztrátovou kompresi. Jak bylo zmíněno, GRP transformace představuje parametrickou verzi Burrows-Wheelerovy transformace rozšířením permutační metody používané v RadixZipu. Tato transformace pak provádí manipulaci s bloky řetězců, které jsou následně lexikograficky tříděny, což se ve výsledném řetězci projevuje výskytem stejných, po sobě jdoucích znaků (stejnou vlastností disponuje výše zmíněná BWT). GRP transformace konvertuje vstupní řetězec (I) o délce n na jiný řetězec (O) o délce n a přirozené číslo L. Kromě vstupního řetězce má GRP transformace další dva vstupní parametry. První z nich reprezentuje velikost bloku, označenou znakem l. Druhým parametrem je pak kontextové uspořádání, označené znakem d. Pro tyto parametry platí dvě vstupní podmínky. První z nich požaduje to, aby velikost bloku (l) byla větší než velikost kontextového uspořádání (d). Druhá podmínka definuje novou proměnnou (b), kde násobek b a l musí odpovídat n.

Tabulka 2: Parametry GRP transformace

Hodnota	Označení	Podmínka
Vstupní řetězec	I	-
Výstupní řetězec	O	-
Délka řetězce	n	lb
Pořadové číslo klíčového sloupce	L	-
Velikost bloku	l	$l > d$
Kontextové uspořádání	d	$d < l$
Rozměr matice	b	lb

Z takto zvolených parametrů jsme následně schopni sestavit matici (5) a do takto vytvořené matice (6) pak můžeme jednoduše přepsat vstupní řetězec.

$$A = (a_{ij})_{(l+d)\times b} \tag{5}$$

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,b-1} & a_{1,b} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,b-1} & a_{2,b} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{l,1} & a_{l,2} & \cdots & a_{l,b-1} & a_{l,b} \\ a_{l+1,1} & a_{l+1,2} & \cdots & a_{l+1,b-1} & a_{l+1,b} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{l+d,1} & a_{l+d,2} & \cdots & a_{l+d,b-1} & a_{l+d,b} \end{bmatrix} \tag{6}$$

3.1.1 Popis dopředné části GRP transformace

Dopředná GRP transformace se skládá z celkových tří kroků. V první části této podkapitoly bude představen pseudokód možné implementace, v druhé části bude každý krok podrobně popsán na adekvátním vstupním řetězci.

Pseudokód:

Algoritmus 1: Dopředná GRP transformace

```
1  function GrpForwardTransform (I, l, d)
   Vstup  : Řetězec I a parametry GRP transformace l a d
   Výstup: Řetězec O a klíčová hodnota L
2  Inicializace:
3      Převeď I o délce n na matici A = (a_ij)_{(l+d)×b}
4      Nastav hodnotu L = b − 1
5  Blokové třídění I:
6      for i = 0; i < d; do
7          Setřiď sloupce v matici A podle hodnot v i-tém řádku matice A
8          Nastav hodnotu L na současnou pozici klíčového vektoru v
9      end
10 Blokové třídění II:
11     for i = d; i < d + l; do
12         Přidej ke stávajícímu výstupu i-tý řádek matice A
13         if i = l + d − 1 then
14             break
15         Setřiď sloupce v matici A podle hodnot v i-tém řádku matice A
16     end
```

3.1.2 Ukázka dopředné části GRP transformace

V této části si postupně ukážeme celý proces dopředné transformace na jednoduchém příkladu.

Krok číslo 1 – Inicializace

V rámci jednoduchého příkladu můžeme uvažovat tento vstupní řetězec (7). Pro takto zvolený vstupní řetězec jsou pak parametry následující $l = 3, d = 2, b = 5, n = 15$.

$$I = \begin{bmatrix} h & o & t & s & p & o & t & s & t & o & p & s & h & o & t \end{bmatrix} \tag{7}$$

Čtvercová matice A bude pro zvolený příklad vytvořena o velikosti $A = (a_{ij})_{(3+2)\times 5}$. Pro ještě lepší představu je zde uvedena i doplněná matice A. (8).

$$A = \begin{bmatrix} h & s & t & o & h \\ o & p & s & p & o \\ t & o & t & s & t \\ s & t & o & h & h \\ p & s & p & o & o \end{bmatrix} \tag{8}$$

Na příkladu číslo (9) lze vidět vliv parametrů na rozložení matice. Počet sloupců je definován parametrem b, počet řádku matice je roven $l + d$. Ve speciálních případech může parametr l pokrýt všechny řádky ($d = 0$).

$$\begin{array}{c} \\ l \\ \\ \\ d \end{array} \overbrace{\begin{bmatrix} h & s & t & o & h \\ o & p & s & p & o \\ t & o & t & s & t \\ s & t & o & h & h \\ p & s & p & o & o \end{bmatrix}}^{b} \tag{9}$$

Takto vyplněná matice pak odpovídá vstupnímu řetězci, kde pro první sloupec matice platí (10) a pro druhý sloupec platí (11).

$$\begin{bmatrix} \overbrace{h}^{l} & o & t & \overbrace{s}^{d} & p & o & t & s & t & o & p & s & h & o & t \end{bmatrix} \tag{10}$$

$$\begin{bmatrix} h & o & t & \overbrace{s}^{l} & \overbrace{p}^{d} & o & t & s & t & o & p & s & h & o & t \end{bmatrix} \tag{11}$$

Do prvního sloupce nejprve umístíme první tři znaky ($l = 3$) ze vstupního řetězce. První sloupec doplníme následujícími dvěma znaky ($d = 2$), čímž jsme doplnili celý první sloupec.

Pro druhý sloupec pokračujeme obdobně, začínáme však u čtvrtého znaku ($1l + 1 = 4$), kde první číslo reprezentuje index sloupce (index začíná na nule).Takto postupně pokračujeme až do naplnění celé matice. Na závěr tohoto kroku musíme nastavit hodnotu $L = b - 1$, v našem případě $L = 4$. V tomto stavu je inicializace hotová a můžeme začít se samotnou transformací (krok číslo 2).

Krok číslo 2 – Blokové třídění I.

Druhý krok vychází z již provedeného prvního kroku (8). V rámci této části jsou tříděny řádky od 0 do d (v našem případě $d = 2$), přičemž toto třídění je aplikováno na jednotlivé sloupce matice.

Pro $i = 0$ třídíme abecedně první (nultý) řádek matice A. V závislosti na tomto třídění přesouváme celé sloupce matice. Tato operace je nastíněna na příkladu číslo (12). Současně musíme brát i ohled na náš klíčový sloupec, nebo spíše jeho pořadové číslo L. Klíčový sloupec se v rámci první iterace přesunul z páté na druhou pozici, a proto musíme nastavit $L = 1$.

$$A = \begin{bmatrix} h & h & o & s & t \\ o & o & p & p & s \\ t & t & s & o & t \\ s & h & h & t & o \\ p & o & o & s & p \end{bmatrix} \tag{12}$$

Pro $i = 1$ třídíme abecedně druhý (první) řádek matice A. Stejně jako v předchozím případě musíme seřadit všechny sloupce matice podle setříděného druhého řádku, a současně také sledovat, na kterou pozici byl umístěn vektor L. V tomto případě je však už druhý řádek abecedně seřazen, a proto není nutné provádět jakékoliv změny. Stejně tak hodnota L zůstane nezměněna.

Krok číslo 3 – Blokové třídění II. a generování výstupu

Třetí krok dále pokračuje v blokovém lexikografickém třídění podle zvoleného klíče (řádku). V tomto případě ale v jednotlivých iteracích získáváme i výsledné části transformovaného vstupu. Řádky matice jsou tentokrát tříděny od $d = 2$ do $d + l = 5$.

Pro $i = 2$ nejdříve získáme již finální část výsledného řetězce. Tento řetězec je umístěn na i-tém řádku matice A. V našem případě (příklad číslo (12)) tak získáme řetězec $[ttsot]$. Poté pokračujeme stejným způsobem jako v kroku číslo 2, tedy lexikografickým tříděním i-tého řádku. Tím získáme opět upravenou matici A, která je zobrazena na příkladu číslo (13).

$$A = \begin{bmatrix} s & o & h & h & t \\ p & p & o & o & s \\ o & s & t & t & t \\ t & h & s & h & o \\ s & o & p & o & p \end{bmatrix} \tag{13}$$

Pro $i = 3$ pokračujeme jako v předchozím případě. Nejdříve získáme řetězec, který je reprezentován čtvrtým řádkem v matici A. Tento řetězec přidáme k již částečně nalezenému řešení, čímž získáváme řetězec [*ttsotthsho*]. Poté pokračujeme lexikografickým tříděním třetího (čtvrtého) řádku matice A. Setříděná matice je zobrazena na příkladu číslo (14).

$$A = \begin{bmatrix} o & h & t & h & s \\ p & o & s & o & p \\ s & t & t & t & o \\ h & h & o & s & t \\ o & o & p & p & s \end{bmatrix} \tag{14}$$

Pro $i = 4$ stejně jako v předchozích případech získáme další část našeho finálního řetězce [*oopps*]. Současně je ale také splněna podmínka $i = l + d - 1$, a proto už neprovádíme žádné další lexikografické třídění. Řetězec získaný v tomto kroku jednoduše přidáme k již dvěma sestaveným částem finálního řetězce. Tento řetězec (spolu s klíčem $L = 1$) představuje finální podobu naší dopředné transformace a reprezentuje tedy i výstup samotné transformace (15).

$$O = \begin{bmatrix} t & t & s & o & t & t & h & s & h & o & o & o & p & p & s \end{bmatrix}, L = 1 \tag{15}$$

3.1.3 Popis inverzní části GRP transformace

Inverzní transformace se skládá z celkových šesti kroků. Stejně jako v případě dopředné transformace bude i zde nejdříve uveden pseudokód a následně detailně popsán příklad inverzní transformace. Inverzní transformace obsahuje oproti dopředné transformaci více kroků, je i mírně složitější, zvláště z pohledu maticových operací. Součástí inverzní transformace jsou i další (zatím nepoužívané matice), které mohou být v budoucnu odstraněny a v příkladu slouží víceméně pro názornost a jednoduché pochopení inverzního transformačního procesu. Tyto konkrétní matice byly pak v implementaci vyřazeny (pokud měly vliv na výkonnost algoritmu).

Pseudokód:

Algoritmus 2: Inverzní GRP transformace

function GrpBackwardTransform (O, l, d, L)
Vstup : Řetězec O, parametry GRP transformace (l, d) a pozice klíčového sloupce L
Výstup: Řetězec I

Inicializace:
    Vytvoř matici $S = (a_{ij})_{l \times b}$ a matici $U = (a_{ij})_{b \times b}$
    Vlož do matice S řetězec O podle pravidla $S_{ij} = O_{(i-1)b+j}$, pro které platí $0 \leq i < l, 0 \leq j < b$.
    Zkopíruj poslední řádek z matice S a ulož ho jako poslední řádek do matice U
Třídění na základě klíče:
    for $i = 1;\ i < l;$ **do**
        Setřiď lexikograficky znaky na řádku $l - i - 1$ matice S, pak ho umísti do matice U na řádek $l + d - i - 1$.
        Setřiď sloupce matice U tak, aby řádek na pozici $l + d - i - 1$ odpovídal řádku matice S na pozici $l - i - 1$.
    end
Doplnění matice U:
    Vytvoř matici $V = (a_{ij})_{d \times b}$
    Zkopíruj d řádků matice U do matice V
    Lexikografiky setřiď sloupce matice V použitím jednotlivých řádků jako klíč
    Zkopíruj d řádků matice V do neinicializované části matice U.
Aplikace klíčového sloupce:
    Nechť w reprezentuje sloupec matice U na pozici L.
    Vytvoř matici $T = (a_{ij})_{l+d \times b}$
    Zkopíruj hodnoty w na pozici posledního sloupce matice T
Doplnění matice T:
    for $i = 0;\ i < b - 1;$ **do**
        Ze sloupců matice U, které zatím nebyly zkopírovány do matice T, vyber první sloupec z leva, který má horních d symbolů stejných jako dolních d symbolů vektoru w
        Hodnoty vybraného sloupce zkompíruj do w a do matice T jako sloupec matice na pozici i.
    end
Získání původního řetězce:
    Z matice T získej původní řetězec pomocí:
        for $i = 0;\ i < l;$ **do**
            for $j = 0;\ j < b;$ **do**
                $I_{i+jl} = T_{ij}$
            end
        end

Jak lze na první pohled vidět, je inverzní transformace daleko komplikovanější než transformace dopředná. I přes to však není ve většině případů inverzní transformace výpočetně nároč-

nější. Speciální případ, kdy je inverzní transformace znatelně pomalejší nastává pouze pokud má většina sloupců v pátém kroku rozdílně hodnoty až v poslední části horních d znaků. V tomto případě algoritmus provádí velké množství vzájemných porovnání, než najde správný sloupec patřičné matice. Z výsledků testování je patrné, že se tento problém v reálných datech obvykle nevyskytuje.

3.1.4 Ukázka inverzní části GRP transformace

Výsledkem dopředné transformace byl výstupní řetězec O a pozice klíčového sloupce matice $L = 1$ (15). V této části si postupně ukážeme proces inverzní transformace na původní řetězec (7).

Krok číslo 1 – Inicializace

V rámci prvního kroku je nejdříve nutné vytvořit matici $S = (a_{ij})_{l \times b}$ a matici $U = (a_{ij})_{b \times b}$. V našem případě bude mít tedy matice S tři řádky a pět sloupců. Matice U bude pak čtvercovou maticí o velikosti pět řádku a pět sloupců. Vstupní řetězec (příklad číslo (15)) pak můžeme jednoduše rozdělit na tři podřetězce o pěti znacích a tyto podřetězce dále vložit do matice S (každý podřetězec délky 5 odpovídá právě jednomu řádku matice S). Poslední částí tohoto kroku je překopírování posledního řádku matice S do matice U. Tyto matice jsou pak znázorněny na příkladu číslo (16).

$$S = \begin{bmatrix} t & t & s & o & t \\ t & h & s & h & o \\ o & o & p & p & s \end{bmatrix}, U = \begin{bmatrix} \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ o & o & p & p & s \end{bmatrix} \tag{16}$$

Krok číslo 2 – Třídění na základě klíče

Druhá část postupuje iterativně pro téměř všechny řádky matice S (všechny řádky kromě posledního). Každý vybraný řádek je nejdříve seřazen lexikograficky a následně překopírován do matice U na první dolní neobsazenou pozici. Tento řádek je nakonec znovu setříděn tak, aby nově vzniklý řádek odpovídal původnímu řádku z matice S. Úpravy v matici U jsou pak již prováděny na celé sloupce. Tento proces je zřetelnější z následujícího příkladu. Cyklus je vykonáván od $i = 1$ do l.

Pro $i = 1$ je potřeba nejprve vybrat příslušný řádek z matice S. Tento řádek se nachází na pozici $l - i - 1$ (v našem případě $l - i - 1 = 1$). Jedná se tedy o řádek s řetězcem $[thsho]$. Tento řetězec je pak lexikograficky seřazen, čímž dostáváme řetězec $[hhost]$. Tento řetězec překopírujeme na pozici $b - i - 1$ matice $U(b - i - 1 = 4)$ a seřadíme sloupce matice U tak, aby nově přidaný řádek odpovídal původnímu řetězci $[thsho]$. Tento proces je naznačen na příkladu číslo (17).

$$U = \begin{bmatrix} . & . & . & . & . \\ . & . & . & . & . \\ . & . & . & . & . \\ h & h & o & s & t \\ o & o & p & p & s \end{bmatrix} \rightarrow \begin{bmatrix} . & . & . & . & . \\ . & . & . & . & . \\ . & . & . & . & . \\ t & h & s & h & o \\ s & o & p & o & p \end{bmatrix} \tag{17}$$

V další iteraci ($i = 2$) opakujeme stejný proces jako v první iteraci. V tomto případě překopírujeme nultý (první) řádek matice S na druhý (třetí) řádek matice U. Tento druhý řádek lexikograficky seřadíme. Následně se opět snažíme přetransformovat druhý řádek matice U tak, abychom dostali původní řetězec z matice S, současně ale při záměně pořadí přesouváme celé sloupce (18).

$$U = \begin{bmatrix} . & . & . & . & . \\ . & . & . & . & . \\ o & s & t & t & t \\ t & h & s & h & o \\ s & o & p & o & p \end{bmatrix} \rightarrow \begin{bmatrix} . & . & . & . & . \\ . & . & . & . & . \\ t & t & s & o & t \\ s & h & h & t & o \\ p & o & o & s & p \end{bmatrix} \tag{18}$$

Krok číslo 3 – Doplnění matice U

Třetí část zpětné transformace má za úkol doplnit (v našem případě) horní dva řádky matice U. Nejdříve tedy vytvoříme novou matici $V = (a_{ij})_{d \times b}$, v našem případě to bude matice o velikosti $V = (a_{ij})_{2 \times 5}$. Do nově vzniklé matice V překopírujeme d spodních řádků matice U (19).

$$V = \begin{bmatrix} s & h & h & t & o \\ p & o & o & s & p \end{bmatrix} \tag{19}$$

Pro tuto matici V postupně (odspodu) lexikograficky seřazujeme řádky. Současně ale třídění provádíme opět na celých sloupcových vektorech. Setříděním dostáváme upravenou matici V (20).

$$V = \begin{bmatrix} h & h & o & s & t \\ o & o & p & p & s \end{bmatrix} \tag{20}$$

V poslední části kroku číslo 3 už jen doplníme prázdné řádky matice U řádky z matice V. Tedy v našem konkrétním případě překopírujeme nultý řádek matice V na pozici nultého řádku matice U. Stejným způsobem upravíme i první řádek. Obecně se jedná o d iterací. Finální matice U je pak znázorněna na příkladu číslo (21).

$$U = \begin{bmatrix} h & h & o & s & t \\ o & o & p & p & s \\ t & t & s & o & t \\ s & h & h & t & o \\ p & o & o & s & p \end{bmatrix} \tag{21}$$

Krok číslo 4 – Aplikace klíčového sloupce

V rámci čtvrtého kroku je nutné vytvořit novou matici $T = (a_{ij})_{l+d \times b}$, v našem případě to bude čtvercová matice o velikosti 5×5. Současně také vytváříme nový řetězec w o velikosti $b (b = 5)$. Do řetězce w překopírujeme L-tý sloupec z matice U, tedy ten sloupec, který má shodné pořadové číslo s naším uloženým pořadovým klíčem L. Jelikož naše pořadové číslo klíčového sloupce je $L = 1$, tak do sloupcového vektoru překopírujeme první sloupec z matice U. Řetězec w pak už jen překopírujeme na sloupcovou pozici b v matici T (22).

$$w = \begin{bmatrix} h \\ o \\ t \\ h \\ o \end{bmatrix}, T = \begin{bmatrix} \cdot & \cdot & \cdot & \cdot & h \\ \cdot & \cdot & \cdot & \cdot & o \\ \cdot & \cdot & \cdot & \cdot & t \\ \cdot & \cdot & \cdot & \cdot & h \\ \cdot & \cdot & \cdot & \cdot & o \end{bmatrix} \tag{22}$$

Krok číslo 5 – Doplnění matice T

Pátý krok je posledním krokem, který provádí úpravu nad maticemi. Naším úkolem je doplnit zbylé neobsazené pozice matice T. Tento proces je prováděn iteračně, kde počet iterací bude roven $b-1$. V matici U hledáme takový řetězec, který má horních d znaků shodných se spodními d znaky řetězce w. Matici U procházíme zleva, první takový sloupec matice pak umístíme na sloupcovou pozici i v matici T. Současně nově přesunutý sloupec matice označíme jako nový řetězec w.

V první iteraci ($i = 0$) hledáme takový sloupec matice, který končí na znaky $[ho]$ (to odpovídá posledním dvěma znakům vektoru w, jelikož $d = 2$). Tento sloupec je umístěn v matici U hned na nulté pozici. Nultý sloupec v matici U tedy překopírujeme na nultou pozici v matici T a stejný sloupec nastavíme jako řetězec w. Stav po tomto provedeném kroku je zobrazen na příkladu číslo (23).

$$w = \begin{bmatrix} h \\ o \\ t \\ s \\ p \end{bmatrix}, T = \begin{bmatrix} h & \cdot & \cdot & \cdot & h \\ o & \cdot & \cdot & \cdot & o \\ t & \cdot & \cdot & \cdot & t \\ s & \cdot & \cdot & \cdot & h \\ p & \cdot & \cdot & \cdot & o \end{bmatrix} \tag{23}$$

Druhá iterace ($i = 1$) probíhá stejně jako první. Hledáme takový sloupec z matice U, který má totožné první dva znaky s posledními dvěma znaky z řetězce w. Hledáme tedy sloupec, který začíná na $[sp]$. Tento sloupec je v matici U umístěn na třetí (čtvrté) pozici. Tento sloupec překopírujeme na pozici i v matici T a stejně tak tento sloupec překopírujeme do řetězce w (24).

$$w = \begin{bmatrix} s \\ p \\ o \\ t \\ s \end{bmatrix}, T = \begin{bmatrix} h & s & \cdot & \cdot & h \\ o & p & \cdot & \cdot & o \\ t & o & \cdot & \cdot & t \\ s & t & \cdot & \cdot & h \\ p & s & \cdot & \cdot & o \end{bmatrix} \tag{24}$$

Ve třetí iteraci ($i = 2$) hledáme sloupec z matice U, který má první dva znaky rovné $[ts]$. Takový sloupec se nachází na poslední pozici v matici U. Tento sloupec překopírujeme do matice T na druhou (třetí) pozici a taktéž jej překopírujeme do řetězce w (25).

$$w = \begin{bmatrix} t \\ s \\ t \\ o \\ p \end{bmatrix}, T = \begin{bmatrix} h & s & t & \cdot & h \\ o & p & s & \cdot & o \\ t & o & t & \cdot & t \\ s & t & o & \cdot & h \\ p & s & p & \cdot & o \end{bmatrix} \tag{25}$$

Ve čtvrté iteraci ($i = 3$) hledáme sloupec z matice U, který začíná na $[op]$. Takový sloupec je umístěn na druhé (třetí) pozici v matici U. Tento sloupec překopírujeme do matice T na poslední neobsazenou pozici. Finální matice T je zobrazena na příkladu číslo (26). V tomto případě již není potřeba kopírovat sloupec matice U do řetězce w, protože je splněna ukončující podmínka $i = b - 1$.

Jak již bylo dříve uvedeno, tento krok je poměrně náročný a to zvláště v případě, kdy jsou rozdílné výhradně poslední znaky z rozsahu, který je definován parametrem d. Za vhodný příklad lze například považovat po sobě jdoucích 100 000 slov „bananna“. V tomto případě budou sloupce matice U a T velice podobné, čímž bude inverzní transformace mnohonásobně náročnější.

$$T = \begin{bmatrix} h & s & t & o & h \\ o & p & s & p & o \\ t & o & t & s & t \\ s & t & o & h & h \\ p & s & p & o & o \end{bmatrix} \tag{26}$$

Krok číslo 6 – Získání finálního řetězce

Matice T zobrazená na příkladu číslo (26) je shodná s maticí A (8), kterou jsme používali při dopředné transformaci. Z této matice už jsme schopni jednoduchým způsobem získat původní řetězec. Jednotlivé sloupce matice T procházíme od 0 do l a z těchto znaků postupně skládáme náš řetězec. Výsledkem je původní řetězec (7).

3.2 Třídící algoritmus Radixsort

Ve výše zmíněné GRP transformaci existuje hned několik kroků, které vyžadují lexikografické třídění. V rámci implementace GRP transformace lze použít libovolný stabilní třídící algoritmus. Stabilní třídící algoritmus je takový algoritmus, který při porovnávání stejných prvků zachovává jejich pořadovou posloupnost. Pro samotné třídění byl zvolen velice výkonný třídící algoritmus Radixsort. Bohužel, jakýkoliv zvolený třídící algoritmus musí být dále upraven tak, aby splňoval podmínky dané transformace. Zvolený Radixsort musíme tedy upravit tak, abychom byli schopni při samotném třídění sledovat pozici klíčového sloupce (L) v patřičné matici. Třídění musí také probíhat na základě zvoleného řádku v matici. Prvkem třídícího algoritmu tak není jeden znak nebo číslo, ale sloupec (řetězec) čísel nebo znaků. Právě z tohoto důvodu je potřeba využití stabilního třídícího algoritmu. Pokud bychom použili nestabilní třídící algoritmus, nemusela by výsledná transformace fungovat. Představme si situaci, kdy lexikograficky třídíme sloupce v matici na základě specifikovaného klíče (řádku). Současně jsou v tomto řádku dva stejné prvky, které mají však rozdílné hodnoty ve sloupcích. Pokud jsou tyto prvky seřazeny lexikograficky, přičemž je jejich vzájemná pozice zaměněna, bude GRP transformace neúspěšná. Přesněji, selže transformace inverzní, která nedokáže se 100 % obnovit vstupní data.

Radixsort představuje stabilní třídící algoritmus, který používá velice zajímavý přistup k třídění dat. Na rozdíl od téměř standardního přístupu, kdy třídící algoritmy různými způsoby porovnávají jednotlivé prvky mezi sebou, je Radixsort založen na principu seskupování podle jednotlivých klíčů. V rámci Radixsortu jsou nejdříve vytvořeny struktury s pořadovými čísly $0 - 9$. Následně postupně procházíme množinu vstupních prvků, které umisťujeme do jednotlivých struktur. Každý prvek je umístěn do takové struktury, která má shodné pořadové číslo s jednotkou právě vkládaného prvku. Tímto způsobem jsou všechny vstupní prvky rozděleny mezi jednotlivé struktury. Poté stačí postupně všechny struktury projít a z prvků, které jsou v nich umístěny, sestavit setříděnou posloupnost. Vkládání do patřičných struktur funguje

na principu FIFO (first in, first out), čímž je zajištěna stabilita třídícího algoritmu. V tomto stavu jsou prvky setříděny, nicméně setříděny jsou pouze podle jednotek. Pokud chceme prvky opravdu setřídit, musíme proceduru opakovat stejným způsobem i pro desítky, stovky, tisíce a podobně. Tímto způsobem jsou tedy postupně všechny prvky rozřazovány až do finální setříděné podoby. Radixsort je tak naprosto perfektním třídícím algoritmem pro vstupy, které nabývají nižších hodnot. V našem případě, kdy budeme třídit jednotlivé znaky nebo malá čísla (Byty), tak bude Radixsort velice efektivní. Složitost algoritmu je rovna $O(wn)$, kde w reprezentuje délku vstupního prvku, a n představuje počet prvků. [13]

3.3 Move to front transformace

Move to front transformace (dále jen MTF) reprezentuje další transformační algoritmus pro datovou kompresi. Jak již bylo zmíněno, transformační algoritmy samy o sobě data nekomprimují, ale místo toho pomáhají ostatním algoritmům při celkové kompresi konkrétních dat. MTF je na základě této skutečnosti obvykle používána ještě před entropickým kódováním. Transformace je taktéž dostatečně výkonná na to, abychom ji v našem řešení zahrnuli. Hlavní myšlenkou MTF transformace je reorganizace vstupních dat na základě patřičné abecedy, přičemž se frekventovanější symboly vyskytují blíže k začátku abecedy. Nejdříve je vytvořena abeceda obsahující všechny symboly, které se vyskytují ve vstupních datech. Pokud máme vytvořenou abecedu, můžeme začít s postupným procházením vstupních dat. Vstupní data jsou iteračně procházena symbol po symbolu, následně jsou všechny výskyty nahrazovány indexem daného znaku z abecedy. Tento znak je dále v abecedě vyhledán a umístěn na začátek. Tímto způsobem jsou procházeny všechny znaky ze vstupních dat. Pořadí znaků v abecedě se tak s každým po sobě jdoucím rozdílným znakem mění. Tento proces zajišťuje výskyt frekventovanějších znaků na začátku abecedy, přičemž méně frekventované znaky mají z pravidla vyšší hodnotu. Vyšší hodnota je zajištěna větším indexem v abecedě (ostatní znaky odsunují tento znak na vyšší index při přesunu na začátek abecedy). MFT algoritmus je lokálně adaptivní. To je zajištěno již zmíněnou automatickou úpravou používané abecedy (úpravou indexů). Existují i jiné varianty MTF transformace. Může se jednat například o transformaci, která pouze odkazuje na indexy ve vytvořené abecedě. Tato abeceda může být pro příklad dále seřazena na základě frekvenčních výskytů jednotlivých znaků. [6]

Představme si příklad se vstupním řetězcem „wawttsotthshooopps“ a vstupní abecedou „tsohp“. Abeceda je zkonstruovaná na základě všech obsažených znaků ve vstupních datech. První testovaný příklad využívá variantu, jež nepoužívá move to front krok, při kterém posouváme právě načtený symbol na začátek abecedy. Tato varianta má průměrnou hodnotu $3,11$, což je velice dobrý výsledek. Bohužel tato varianta je naprosto nevhodná pro větší vstupy a větší abecedy, kde následně průměrná velikost roste mnohonásobně oproti ostatním variantám. Druhá varianta je velice podobná té první. Na začátku je však vstupní abeceda seřazena podle počtu výskytů jednotlivých znaků. U druhé varianty je průměrná hodnota $2,22$, což představuje znatelné zlepšení oproti první variantě. Druhá varianta je určitě efektivnější, protože používá

seřazenou abecedu, která zachycuje statistické údaje o samotných datech. Třetí varianta již přesouvá právě načtený znak na začátek abecedy. Tato varianta dosáhla průměrné hodnoty $1,88$, což představuje ještě další zlepšení oproti druhé variantě. U třetí metody lze taktéž vidět adaptivní chování. Výstupní abeceda je $A = [s, p, o, h, t, w, a]$. Na začátku iteračního procházení se vyskytuje znak ‚a', který se dále ve vstupním řetězci nevyskytuje. Po dokončení transformace je tento znak umístěn na posledním místě v abecedě právě proto, že všechny ostatní znaky byly posunuty před znak ‚a'. Stejně tak můžeme vidět, že je znak ‚w' umístěn na předposlední místo. Ten se sice vyskytoval po znaku ‚a', ale pouze jednou. Posledním znakem ve vstupním řetězci byl znak ‚s'. Ten je ve výstupní abecedě umístěn na prvním (nultém) místě, protože se algoritmus snaží adaptivně přizpůsobit právě načtenému znaku. Čtvrtá varianta je kombinací dvou předchozích variant. Vstupní abeceda je seřazena na základě počtů výskytů jednotlivých znaků. Poté následuje, stejně jako ve třetí variantě, postupné procházení a přesouvání právě načteného znaku na začátek abecedy. Čtvrtá varianta má průměrnou hodnotu $2,38$, což představuje mírné zhoršení oproti třetí variantě. Čtvrtá varianta bude obecně lépe fungovat na menších vstupech, kde třetí varianta nemá dostatek času na adaptaci. Konkrétní výsledky všech variant jsou zaznačeny v tabulce číslo 3. Z výsledků můžeme usuzovat, že je na tomto vstupu třetí varianta nejefektivnější. Díky adaptivnímu přístupu pak na reálných datech funguje nejlépe právě třetí varianta. Na druhou stranu lze určitě sestavit příklad, kde bude první, druhá nebo čtvrtá varianta nejlepší. Tyto příklady však obvykle neodráží skutečnou reprezentaci reálných dat, a proto bude v kompresní knihovně využita třetí varianta.

Tabulka 3: Výsledky variant MTF transformace

Varianta	Vstupní abeceda	Výstup MTF transformace	H	$\overline{x}$
1.	$[w, a, t, s, o, h, p]$	$[0, 1, 0, 2, 2, 3, 4, 2, 2, 5, 3, 5, 4, 4, 4, 6, 6, 3]$	$2,68$	$3,11$
2.	$[t, o, s, w, h, p, a]$	$[3, 6, 3, 0, 0, 2, 1, 0, 0, 4, 2, 4, 1, 1, 1, 5, 5, 2]$	$2,68$	$2,22$
3.	$[w, a, t, s, o, h, p]$	$[0, 1, 1, 2, 0, 3, 4, 2, 0, 5, 3, 1, 3, 0, 0, 6, 0, 3]$	$2,49$	$1,88$
4.	$[t, o, s, w, h, p, a]$	$[3, 6, 1, 2, 0, 4, 4, 2, 0, 5, 3, 1, 3, 0, 0, 6, 0, 3]$	$2,64$	$2,38$

3.4 Run-Length Encoding

RLE (run-length encoding) reprezentuje jednoduchou kompresní metodu, která přepisuje posloupnost stejných prvků jako jsou například znaky do trojice $< i, l, e >$, kde znak i reprezentuje identifikační znak, l reprezentuje délku posloupnosti a znak e daný prvek. Tato kompresní metoda je však značně závislá na charakteru vstupních dat. Tento typ komprese se dá využít velice dobře na datech, která obsahují stejné posloupnosti určitých prvků. Může se jednat například o posloupnosti stejných znaků nebo pixelů. Stejné, po sobě jdoucí znaky se ale v běžném textu normálně nevyskytují. To není ale jediný problém, který může nastat při RLE kompresi. Při používání této metody je nezbytně nutné definování způsobu zápisu. Představme si kompresi textového dokumentu. Pokud postupně procházíme všechny znaky, které přepisujeme do

trojice identifikující posloupnosti stejných znaků, pak můžeme dostat daleko větší výstup, než byl původní vstup. To je samozřejmě nežádoucí, a proto je potřeba zohlednit tyto problémy při implementaci. Na následujících příkladech je vysvětleno, proč je potřeba použít identifikační trojici. [6]

Pro vstupní řetězec „AAAAAABBBBAAABBBBB“ bude výstupním řetězcem „6A4B3A5B“. Jednoduchým průchodem jsme vždy spočítali posloupnost stejných znaků a na konci této posloupnosti jsme ji dále zapsali do výstupního řetězce ve tvaru $< l, e >$. Tento přístup má však hned několik nedostatků. Pokud bychom stejnou metodu použili na vstupní řetězec „ABCCDE“, dostaneme výstupní řetězec „1A1B2C1E1D“. Tím jsme ale velikost výstupu razantně zvýšili. Proto se nabízí jednoduchá úvaha, která bude přepisovat do již dříve zmiňované dvojice pouze posloupnosti délky 2 a více. Nicméně tady se setkáváme s dalším problémem. Do této chvíle nebyl objasněn princip dekódování. Ten je velice intuitivní a skládá se z postupného procházení zakódovaných dat. Jednoduchým nahrazením všech dvojic $< l, e >$ patřičnou posloupností dostaneme zpětně původní data. Problém však nastává, pokud původní data obsahují číselné hodnoty. Představme si použití výše zmíněné metody na řetězec „AB2CCCCCDD111AA“. Pro takový vstup bude výstupní řetězec roven „AB25C2D312A“. To je ale výstup, z kterého není jasné, která část řetězce reprezentuje komprimovanou dvojici a která část nekomprimovaný znak čísla. Řešení tohoto problému již bylo nastíněno v úvodu této kapitoly, a tím je použití trojice $< i, l, e >$. Identifikační znak označuje začátek komprimované části. To jednoznačně zajišťuje správnou dekompresi zakódovaných vstupních dat. Podmínkou pro tento způsob RLE komprese je využití takového identifikačního znaku, který se v daných datech nevyskytuje. [6]

3.5 Huffmanovo kódování

Huffmanovo kódování představuje jeden z nejznámějších statistických kompresních algoritmů. Používá se v celé řadě známých aplikací, přičemž některé z nich využívají Huffmanovo kódování jako plnohodnotnou kompresní metodu. Ostatní aplikace pak využívají toto kódování jako jednu z použitých metod v komplexním kompresním procesu. Kódování navrhl David A. Huffman při svých studiích na MIT v rámci semestrálního projektu. Algoritmus byl dále publikován v článku z roku 1952 [14]. Huffmanovo kódování je velice podobné tomu, které navrhli Shannon Claude a Robert Fano. Na rozdíl od Shannon-Fanova kódování produkuje obecně Huffmanovo kódování lepší výsledky. Obě metody pak fungují nejlépe, pokud jsou pravděpodobnosti vstupních symbolů rovny záporným mocninám dvojky. Největší rozdíl mezi těmito metodami spočívá v přístupu při konstrukci binárního stromu. Shannon-Fanovo kódování vytváří strom z jeho vrcholu a postupuje iteračně k jeho spodním částem (přístup shora-dolů). Na druhou stranu Huffmanovo kódování využívá opačného procesu a vytváří binární strom ze spodních částí (listů) a postupuje směrem nahoru až ke kořeni (přístup zdola-nahoru). Huffman při svém testování definoval dvě základní vlastnosti, které se týkají optimálních prefixových kódů:

- V optimálním kódu mají symboly s větší pravděpodobností výskytu kratší kódová slova než symboly s nižší pravděpodobností výskytu.
- V optimálním kódu mají dva symboly, které se nejméně vyskytují ve vstupních datech, stejně dlouhou délku.

Huffmanovo kódování bylo vytvořeno právě na základě těchto dvou vlastností. Tedy vlastností, kdy se dva symboly s nejmenší pravděpodobností výskytu liší pouze v posledním bitu. [5, 6]

Huffmanovo kódování reprezentuje algoritmus využívající proměnnou délku zápisu. Proměnná délka zápisu je určována na základě kódovací tabulky. Ta je sestavena z vytvořeného binárního stromu, kde každý list reprezentuje výskyt jednoho symbolu. Současně však platí, že symboly s menší pravděpodobností výskytu mají delší bitovou posloupnost než symboly s vyšší pravděpodobností výskytu. Sestavení binárního stromu pro Huffmanovo kódování postupuje následovně:

1. Procházíme postupně celá vstupní data, na základě pravděpodobnosti výskytu jednotlivých symbolů sestavíme seřazený seznam symbolů.
2. Pro každý vytvořený symbol ze seznamu symbolů vytvoříme odpovídající vrchol a ten umístíme do seznamu vrcholů.
3. Pro dva vrcholy s nejmenší pravděpodobností výskytu vytvoříme rodičovský vrchol, kde je pravděpodobnost výskytu rovna součtu pravděpodobností obou potomků.
4. Odstraníme oba vrcholy ze seznamu, který obsahuje zatím nezpracované vrcholy.
5. Pokud obsahuje seznam vrcholů více než jeden vrchol, tak pokračujeme krokem číslo 3.
6. Poslední vrchol reprezentuje kořen stromu.

Následně je potřeba iteračně projít celá vstupní data a ty na základě vytvořeného binárního stromu upravit do výsledné podoby. Jednotlivé listy binárního stromu můžou být dále pro efektivnost umístěny do pole. V tomto případě pak nemusíme procházet celý binární strom v každé iteraci. Nejvíce efektivní strukturou pak pravděpodobně bude hashmapa (hashset), který je ve většině programovacích jazyků výkonnější než list nebo pole s více jak 20 prvky. Konkrétní způsob implementace však bude rozebrán až v následující kapitole. Poslední část komprese reprezentuje uložení používaného binárního stromu, ten je potřeba uložit pro následnou dekompresi. Dekomprese se skládá z celkových dvou částí. Nejdříve je potřeba znovu sestavit binární strom. Poté jsou komprimovaná data procházena iteračně bit po bitu. S každým dalším bitem se dostáváme do jiného vrcholu v binárním stromě, pokud dorazíme do listu, tak jsme získali dekomprimovaný symbol (nebo například Byte). [14, 15]

3.6 Splay Tree komprese

Splay tree reprezentuje adaptivní datovou strukturu, která vychází z principu binárního vyhledávacího stromu. V této stromové struktuře proto platí, že každý uzel má maximálně dva a minimálně jednoho potomka (pokud se nejedná o list). Každý uzel má dále v levém podstromu vždy potomky s menší hodnotou a v pravém podstromu vždy potomky s větší hodnotou. Splay tree však tuto funkčnost dále rozšiřuje o adaptivní chování. To je prováděno reorganizací ve stromě na základě právě přečteného uzlu. V případě, kdy přečteme specifický uzel z této stromové struktury, je daný uzel přesunut do kořene celého stromu. Současně jsou však zachovány všechny výše zmíněné vlastnosti binárního vyhledávácího stromu. Reorganizace stromu je prováděna pomocí předem definovaných rotací. Tyto rotace jsou obvykle označovány slovem *Splay*. Rotace jsou prováděny při jakékoliv operaci nad daným stromem. Pokud například vkládáme nový uzel do stromu, je tento uzel vložen na pozici, která je definována podle pravidel binárního vyhledávacího stromu. Poté je tento uzel pomocí operace *Splay* postupnými rotacemi přesunut do kořene stromu. Benefitem tohoto přístupu je právě adaptivní chování, které postupnou reorganizací posouvá více frekventované uzly blíže ke kořenu. [16, 17]

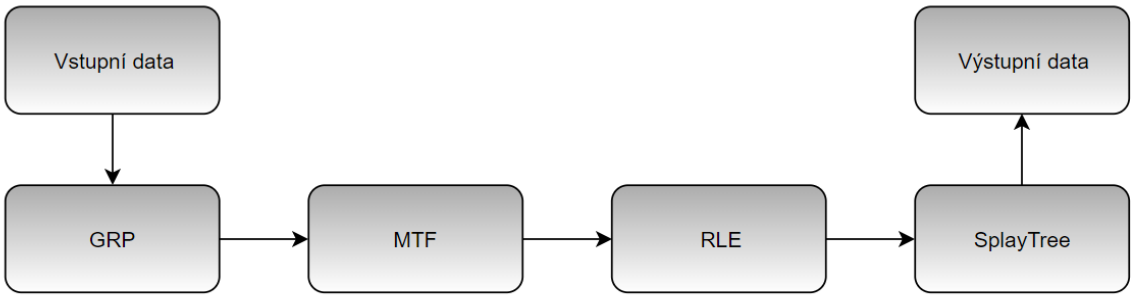
Splay tree komprese je založena na této stromové struktuře. Jednotlivým uzlům jsou přiřazeny hodnoty, přičemž tyto hodnoty mohou být reprezentovány jednotlivými chary nebo Byty. Při vyhledávání jsou postupným procházením generovány jednotlivé bity na výstup. V případě, kdy při vyhledávání postupujeme do levého potomka, zapisujeme na výstup bitovou nulu, při postupu do pravého potomka zapisujeme bitovou jedničku. Tímto způsobem tak generujeme binární posloupnost, která reprezentuje daný char nebo Byte. Tato komprese je v souvislosti s GRP transformací velice účinná. Pokud máme posloupnost stejných Bytu, je u prvního Bytu provedena rotace, která tento Byte přesouvá do kořene stromu. V případě každého dalšího stejného Bytu je pak tento Byte umístěn přímo v kořenu a má takto přiřazenou nejkratší možnou posloupnost bitu.

4 Popis implementace

Cílem implementace bylo vytvoření experimentální aplikace pro bezeztrátovou kompresi. Celá implementace byla rozdělena do dvou hlavních části. První část reprezentuje C# knihovna (DLL) obsahující kompresní algoritmy a několik dalších tříd, které jsou při kompresi použity. Druhou část představuje konzolová aplikace, která využívá vytvořenou knihovnu k plnohodnotné kompresi. Konzolová aplikace umožnuje komprimovat a dekomprimovat soubory s použitím rozdílných parametrů, čímž demonstruje další použití knihovny. Všechny kompresní algoritmy jsou navrženy a naimplementovány tak, aby podporovaly paralelní zpracování dat. Současně s implementační fází probíhalo i neustálé testování a optimalizace jak nově napsaného kódu, tak i kódu původního.

4.1 Kompresní algoritmy a jejich použití

V kapitole číslo 3 byly popsány použité algoritmy pro finální kompresní aplikaci. Mezi tyto algoritmy patří GRP transformace, Radixsort, Run-Length Encoding, Move to front transformace, entropické kódování pomocí Splay Tree a Huffmanovo kódování. Pořadí použitých kompresních algoritmů je nastíněno na obrázku číslo 1.



Obrázek 1: Pořadí kompresních metod

V prvním kroku potřebujeme načíst vstupní data. Vstupní data načítáme do paměti, kde jsou následně uloženy jako pole Bytu. Poté jsou vhodně zvoleny parametry a pole Bytu je předáno GRP transformaci. Data jsou transformována, přičemž výstupní pole Bytu má nepatrně větší velikost než pole vstupní. Výstupní pole Bytu obsahuje navíc kontrolní součet a výsledný parametr L, který je potřebný k inverzní GRP transformaci. Pole Bytu je dále předáváno Move to front transformaci. Ta danou posloupnost Bytu opět proudově transformuje a mění tak entropii samotných dat. Byty jsou poté předány předposlednímu algoritmu RLE. Run length encoding už data komprimuje a do nich navíc přidává pouze dva Byty. První z nich označuje identifikační znak použitý při RLE kompresi (objasněno v kapitole číslo 3.4). Druhý Byte reprezentuje rozdělující znak, který identifikuje konec daného bloku. Poslední kompresní část reprezentuje entropické kódování pomocí Splay Tree algoritmu. Čtvrtý krok může být nahrazen Huffmano-

vým kódováním, které v některých případech produkuje lepší výsledky než Splay Tree kódování. Huffmanovo kódování je naneštěstí v některých souborech znatelně horší. Z tohoto důvodu je ve čtvrtém kroku použito adaptivní kódování pomocí Splay Tree algoritmu. Výsledná data jsou na základě zvolené cesty zapsána do souboru. Určování parametrů a dalších podmínek je rozebráno v konkrétní podkapitole implementace pro každý kompresní algoritmus.

4.2 Kompresní knihovna

Kompresní knihovna obsahuje hned několik souborů, přičemž jeden soubor může obsahovat jednu a více tříd, které souvisejí s daným kompresním algoritmem. Implementace knihovny se opírá o standard objektově orientovaného programování s využitím paralelního (vícevláknového) zpracování. Kompresní knihovna obsahuje následující soubory:

- FileHandler.cs
- GrpTransform.cs
- MtfTransform.cs
- RunLengthEncoding.cs
- SplayTree.cs
- CompressionPack.cs
- Utility.cs

4.2.1 Třída FileHandler

Jak již název napovídá, `FileHandler` reprezentuje třídu, která obstarává práci s diskem. Třída obsahuje metody pro kontrolu absolutní a relativní cesty. Pomocí těchto metod lze jednoduše ověřit, zda zadaná lokace vůbec existuje a jestli odkazuje na cílový soubor nebo složku, což je vhodné při kontrole vstupních argumentů v aplikaci. Třída dále obsahuje metody pro čtení ze souboru a zápis do souboru (string, pole Bytu).

4.2.2 Třída GrpTransform a GrpParameter

Třída `GrpTransform` obsahuje metody pro práci s GRP transformací. Třída má definovaných hned několik vlastností a metod, kde každá z nich hraje určitou roli v transformačním procesu. Celá třída `GrpTransform` obsahuje 7 vlastností, kde každá z nich značně ovlivňuje výpočet a průběh celé transformace. Mezi definované a libovolně volitelné vlastnosti patří:

- PathToParametersTable - Cesta k souboru, ve kterém jsou definovány parametry pro transformaci. Parametry jsou seřazeny podle velikosti vstupu.

- ParametersFromFile - Boolovská hodnota označující, zda mají být parametry načítány ze souboru.
- ParametersFromMemory - Druhá boolovská hodnota označující, zda mají být parametry načítány přímo z paměti.
- MaxBlockSize - Velikost v Bytech, která označuje maximální možnou velikost bloku transformace.
- BlockSize - Velikost v Bytech označující používanou velikost jednoho bloku.
- AvailableThreads - Počet vláken, které jsou použity při výpočtech. Na začátku je tato hodnota nastavena automaticky na počet virtuálních jader procesoru.
- ParametersMemory - List objektů typu `GrpParameter`, který udržuje vygenerované parametry.

Při dopředném a inverzním transformování provádíme nejdříve kontrolu parametru pro GRP transformaci. Parametry pro samotnou transformaci jsou vybírány automaticky na základě definovaných podmínek a výsledků testování. Parametry transformace jsou uloženy v objektech typu `GrpParameter`. Při dopředném a inverzním transformování provádíme nejdříve kontrolu parametru pro GRP transformaci. Parametry pro samotnou transformaci jsou vybírány automaticky na základě definovaných podmínek a výsledků testování. Parametry transformace jsou uloženy v objektech typu `GrpParameter`. Protože tabulka parametrů obsahuje velké množství záznamů, jsou tyto záznamy uloženy v souboru nebo přímo v paměti a nejsou znovu generovány při každém výpočtu. Vygenerované parametry jsou dále seřazeny podle velikosti vstupu. Při prvním výpočtu transformace (první výpočet po spuštění aplikace) provádíme kontrolu parametrů. Pokud jsou parametry vyhledávány přímo v souboru, je nejprve nutné provést kontrolu datového souboru. Datový soubor může být poškozen nebo nemusí zatím vůbec existovat. Z tohoto důvodu je provedena kontrola, přičemž ověřujeme, jestli datový soubor s parametry vůbec existuje. V případě, kdy datový soubor neexistuje, jsou parametry vygenerovány a uloženy do souboru. Pokud načítáme parametry z paměti, jsou tyto parametry vygenerovány a uloženy do předpřipraveného listu a v paměti jsou uchovávány do ukončení aplikace. Maximální velikost, pro kterou jsou parametry generovány, je definována pomocí vlastnosti *MaxBlockSize*. Načítání parametrů ze souboru je pomalejší než vyhledávání v paměti, nicméně velikost parametrické tabulky je úměrná k maximální velikosti vstupu, pro který jsou tyto parametry generovány. Pokud tedy nastavíme *MaxBlockSize* na 500 MegaBytu a budeme vyhledávat parametry z paměti, máme v paměti přibližně 500 MegaBytu velkou tabulku. To nemusí být vhodné na všech zařízeních. Obvykle je však tato hodnota nastavena maximálně na několik málo desítek MegaBytu a proto je načítání z paměti preferovanou možností. Parametry *BlockSize* a *AvailableThreads* jsou spojeny s paralelním zpracováním vstupních dat. Vzhledem k tomu, že naše transformace

představuje nejnáročnější část celého kompresního procesu, jsou vstupní data rozdělena na jednotlivé bloky o velikosti, kterou definuje vlastnost *BlockSize* (stejnou metodu rozdělení do bloků využívá například i známý kompresní program bzip2 [18]). Pro takto rozdělené bloky dat jsou následně nalezeny parametry GRP transformace. Parametry jsou vyhledávány pro bloky o definované velikosti a pro poslední blok, který je obvykle menší než definovaná velikost bloku. Tyto bloky jsou pak paralelně zpracovány na základě počtu dostupných vláken. Počet vláken je definován vlastností *AvailableThreads*, přičemž vlákna mohou zpracovávat bloky dat v libovolném pořadí. Na základě definovaných indexů provedou vlákna GRP transformaci a výsledek umístí do předem vytvořeného pole. Počet současně běžících vláken je omezen pomocí semaforu. Celý proces je pro přehlednost naznačen na vývojovém diagramu (obrázek číslo 9).

Výše popsaný proces je efektivně rozložen do jednotlivých metod, které slouží k výpočtům GRP transformace. Funkce zahrnují použití třídících algoritmů, tabulky parametrů, bloků a paralelního zpracování. Mezi tyto funkce patří:

- ForwardTransformation - Funkce, která nejdříve zkontroluje, zda existuje vytvořená tabulka parametrů, poté na základě velikosti vstupů získá počet jednotlivých bloků a pro každý z nich vytvoří počáteční a koncový index (který odkazuje na umístění ve vstupních datech). Následně jsou vytvořena vlákna pro zpracovávání bloků. Jedno vlákno zpracovává vždy jeden blok. Před samotným spuštěním jsou z tabulky vybrány potřebné parametry pro GRP transformaci. Vlákna jsou poté postupně spouštěna a počet souběžných vláken je omezován pomocí semaforu. Na konec každého bloku jsou zapsány 4 pomocné Byty (hodnota parametru L a velikost nadbytečných dat). Po dokončení všech vláken jsou k celkové velikosti přičteny ještě kontrolní součty a velikost bloků, které byly použity při transformováni (maximálně 8 Bytu).

- BackwardTransformation - Tato funkce je velice podobná funkci předchozí. Nejdříve probíhá kontrola tabulky parametrů. Poté jsou z transformovaných dat získány kontrolní součty, díky kterým je možné vstup rozdělit na určitý počet bloků. Pro tyto bloky jsou znovu vytvořeny počáteční a koncové indexy. Z jednotlivých bloků jsou získány nadbytečné informace (parametr L a kontrolní součet). Pro velikosti bloků jsou následně získány parametry z tabulky, které dále využijeme při inverzní transformaci. Stejně jako v předchozí funkci jsou nakonec vytvořena vlákna, která postupně spouštíme. V posledním kroku už pouze čekáme na dokončení všech vláken, kdy získáme rekonstruovaná původní data.

- ForwardTransformationByteThread - Funkce, která je zodpovědná za dopřednou část GRP transformace. Tuto funkci volají jednotlivá vlákna. Transformace je zpracovávána podle pseudokódu, který je uveden v kapitole 3.1.1.

- BackwardTransformationByteThread - Funkce zodpovědná za inverzní část GRP transformace. Funkce je opět volána jednotlivými vlákny. Inverzní transformace postupuje podle navrženého pseudokódu v kapitole 3.1.3.

- RadixSort - V celé třídě jsou použity celkem tři funkce třídícího algoritmu Radixsort. Dva z nich přejímají matici a index řádku, podle kterého mají být všechny sloupce lexikograficky setříděny. Rozdíl mezi první a druhou implementací Radixsortu spočívá ve vstupním datovém typu. U prvního Radixsortu jsou použity pouze matice s datovým typem Byte, který nemůže nabývat *null* hodnot. Tato funkce je využita v dopředné části transformace. Druhá verze Radixsortu zastává naprosto stejnou funkci s jediným rozdílem. Parametricky přijímaná matice může obsahovat *null* hodnoty. Tato verze je použita v inverzní části, kde musíme pracovat s *null* hodnotami. Třetí verze pak obsahuje zabudovanou funkcionalitu, která kromě třídění dále monitoruje i pozici klíčového vektoru, který je označen pomocí znaku L.
- SortSimiliarity - Funkce, která provádí setřídění v inverzní částí GRP transformace (Druhý krok – Třídění na základě klíče).
- CreateParameterTable - Funkce vytvářející tabulku parametrů GRP transformace.
- ParametersToMemory - Funkce, která nejdříve vytvoří danou tabulku parametrů a tu následně uloží do předem vytvořeného listu parametrů (uložení parametrů v paměti). Funkce přijímá jako parametr maximální velikost vstupu.
- ParametersToFile - Funkce, která nejdříve vytváří tabulku parametrů a tu následně ukládá do zvoleného souboru. Funkce přejímá dva parametry, a to omezení maximální velikosti vstupu (maximální velikost, pro kterou budou parametry generovány) a absolutní nebo relativní cestu, kde bude daný soubor uložen.
- FindParameters - Funkce slouží k vyhledávání parametrů. Na základě zvolených vlastností a předané velikosti vstupu funkce vrací patřičné parametry GRP transformaci.

Všechny funkce prošly významnou optimalizací, přičemž třída `GrpTransform` může obsahovat i další nezmíněné metody. Tyto metody byly používány v testovací části a do finální aplikace nebyly zařazeny. Jedná se například o metodu ForwardTransformationByte, která nebyla dříve přizpůsobena indexům a sloužila pouze k sekvenčnímu zpracování vstupu. Ve třídě byla tato metoda ponechána právě z důvodu testování.

4.2.3 Třída MtfTransform

Třída `MtfTransform` obsahuje implementované metody pro paralelní zpracování Move to front algoritmu. Třída je implementována podle návrhu v kapitole číslo 3.3 (varianta 3). MTF představuje druhý algoritmus, kterému v kompletním kompresním procesu předchází GRP transformace. Třída obsahuje 4 jednoduché metody, které jsou návrhem velice podobné metodám z GRP transformace. Třída taktéž obsahuje dva libovolné parametry *AvailableThreads* a *BlockSize*. Stejně jako v předchozí třídě označuje vlastnost *AvailableThreads* maximální množství vláken, která mohou být použita při samotném transformování. Vlastnost *BlockSize* označuje

maximální velikost bloků, podle kterých jsou data rozdělena. Bloky pro GRP a MTF transformace nemusí být stejně velké, přičemž rozdělení na bloky v MTF transformaci může být úplně vypnuto.
Z výsledků testování je patrné, že vhodná velikost bloku se pohybuje v rozmezí $2-10$ MegaBytu. Při této velikosti je transformace dostatečně rychlá, přičemž použití vyšší velikosti bloků má už pouze zanedbatelný až nulový vliv na výsledný kompresní poměr. Mezi metody této třídy patří:

- Encode – Metoda, která přijímá jako parametr vstupní data v poli Bytu. Tato data jsou na základě definované velikosti bloku rozdělena a pro každou takovou část jsou pak vytvořeny pozice počátečních a koncových indexů, které odkazují do původního pole dat. Na základě této skutečnosti jsou bloky paralelně zpracovávány. Po paralelním zpracování se čeká na synchronizaci všech vláken. Po dokončení zapisujeme na konec dat kontrolní součet a použitou velikost bloků.

- EncodeThread – Tato metoda reprezentuje práci jednoho vlákna a aplikuje MTF transformaci na vstupní blok. Nejdříve je vytvořena abeceda s rozsahem $0-255$ Bytu. Následně jsou iteračně procházena vstupní data, která jsou transformována do výstupního pole.

- Decode – Metoda, která přijímá transformovaná data a ta dekóduje. Nejdříve jsou z dat získány kontrolní součty, na základě kterých je vypočtena délka výstupních dekódovaných dat a pozice jednotlivých indexů. Pro každý blok definovaný indexy vytváříme vlákno. Tyto vlákna jsou následně spuštěna, přičemž počet současně spuštěných vláken je omezen semaforem.

- DecodeThread – Tato metoda opět reprezentuje práci jednoho vlákna a provádí inverzní MTF transformaci. Výstupem jsou původní data.

4.2.4 Třída RunLengthEconding

Třída `RunLengthEconding` obsahuje implementované funkce pro zpracování kompresní metody RLE. Metoda představuje třetí část celého kompresního procesu a navazuje na provedenou MTF transformaci. Stejně jako předchozí implementace obsahuje i tato třída vlastnosti definující maximální počet současně běžících vláken a velikost bloků. Na rozdíl od GRP transformace je však tato metoda poměrně rychlá i při sekvenčním zpracování. Paralelní zpracování bylo implementováno převážně pro velké soubory, u kterých je rozdíl v době výpočtu už znatelný. Kompresní algoritmus je při paralelním zpracování přibližně o 40 % rychlejší než jeho sekvenční verze. To platí i pro menší soubory. V případě dekompresní části je paralelní zpracování téměř vždy pomalejší než zpracování sekvenční. Při paralelní dekompresi musíme nejdříve daný vstup rozdělit na jednotlivé bloky. Následně je potřeba vytvořit vlákna a těm musíme patřičné bloky přidělit. Tento proces je obvykle náročnější než sekvenční průchod polem, při kterém nahrazujeme trojice (kapitola 3.4) patřičnou posloupností Bytu. Z tohoto důvodu je v této třídě definována

ještě jedna vlastnost, pomocí které lze vynutit sekvenční zpracování dekompresní části. Mezi implementované metody třídy patří:

- Encode – Funkce, která rozděluje vstupní data na jednotlivé bloky. Funkce taktéž vytváří pole vláken, kde každé vlákno zpracovává patřičný blok. Před samotným vytvořením vláken je nalezen unikátní symbol, který používáme pro identifikaci trojice.
- EncodeThread – Funkce, která na konkrétní blok aplikuje kompresní algoritmus.
- Decode – Funkce, která reprezentuje dekompresní algoritmus. Pokud mají být data zpracovávána dekompresním algoritmem sekvenčně, voláme funkci *DecodeSignleThread*, které předáváme vstupní data. V případě, kdy mají být data zpracovávána paralelně, jsou data rozdělena na patřičné bloky, kde pro každý blok vytváříme jedno vlákno. Vlákna následně zpracovávají data paralelně. Po dokončení všech vláken jsou původní data v blocích rekonstruována.
- DecodeThread – Funkce, která na konkrétní blok aplikuje dekompresní algoritmus.
- DecodeSingleThread – Funkce, která na celá data aplikuje dekompresní algoritmus. Data jsou zpracovávána pouze sekvenčně.

4.2.5 Třída SplayTree a SplayTreePack

Třída `SplayTree` obsahuje implementovanou stromovou strukturu Splay Tree, která byla objasněna v kapitole číslo 3.6. Tato metoda představuje poslední část kompresního procesu a navazuje binárním kódováním na RLE. Vzhledem k efektivní implementaci byla tato třída implementována pouze sekvenčně. Třída obsahuje následující funkce:

- Initialize – Při vytvoření instance této třídy je nejdříve v konstruktoru předána velikost abecedy. Následně jsou vytvořena tři pole, které reprezentují celý binární vyhledávací strom. Následně je volána tato funkce, která do patřičného stromu dosadí hodnoty na základě předem definované velikosti abecedy.
- Splay - Funkce, která provádí rotace nad binárním vyhledávacím stromem. Tato funkce je volána vždy, když vyhledáme a přečteme konkrétní Byte.
- GetCode - Funkce, která postupně prochází binární vyhledávací strom. V případě, kdy postupujeme do levého potomka, je na výstup předáván bit s nulovou hodnotou. V opačném případě, kdy postupujeme do pravého potomka, předáváme na výstup jedničku. Pokaždé, kdy vygenerujeme určitou binární posloupnost, je provedena funkce *Splay*.
- GetByte - Funkce, která na základě příchozích bitu prochází binární vyhledávací strom. V případě bitové nuly postupujeme do levého potomka. Pokud právě čtená hodnota nabývá bitové jedničky, postupujeme do pravého potomka. Stejně jako v případě předchozí metody je po každém nalezeném Bytu provedena funkce *Splay*.

- Třída obsahuje i další funkce *ByteToIndex* a *IndexToByte*, které pouze odkazují na dané pozice ve vytvořených polích.

Vytvořená třída `SplayTreePack` dále objektově zapouzdřuje třídu předchozí. Jsou zde funkce, které nejdříve vytvoří instanci třídy `SplayTree` s definovanou velikostí abecedy. Nad tímto objektem jsou dále iteračně volány metody *GetCode* nebo *GetByte* (v závislosti na tom, jestli právě komprimujeme nebo dekomprimujeme). Vrácené hodnoty těchto funkcí jsou následně zapisovány na výstup.

4.2.6 Třída CompressionPack

Třída `CompressionPack` spojuje všechny implementované algoritmy v jeden souhrnný proces a umožňuje provádět plnohodnotnou kompresi a dekompresi jednoduchým voláním jedné z metod. Třída obsahuje vytvořené instance všech výše zmíněných tříd. U těch jsme schopni pomocí přetíženého konstruktoru nastavit libovolné parametry pro jednotlivé kompresní algoritmy (velikost bloků, maximální počet používaných vláken a podobně). Stejně tak můžeme použít bezparametrický konstruktor k nastavení výchozích hodnot, které byly doporučeny na základě výsledků testování.

4.2.7 Třída Utility

Třída `Utility` obsahuje pomocné funkce, které mají obvykle informativní charakter. Pomocí této třídy jsme schopni jednoduše spočítat kompresní poměr, kompresní faktor a Entropii nad určitými daty (soubor, pole Bytu, list Bytu a podobně). Třída taktéž obsahuje metody pro generování MD5 hash hodnoty, která byla použita pro porovnávání původních a dekomprimovaných souborů. Dále jsou zde obsaženy tři metody pro výpis velikosti souboru nebo vstupního pole v zadaných jednotkách (bit, Byte, KiloByte, a podobně). Ty mají pouze informativní charakter a jsou použity pro výsledné porovnávání kompresního poměru. Ve třídě jsou obsaženy i další informativní funkce jako je například výpis 2D pole do konzole.

4.3 Kompresní experimentální aplikace

Kompresní experimentální aplikace využívá výše popsanou kompresní knihovnu. Tato konzolová aplikace demonstruje jednoduché použití kompresní knihovny a umožňuje kompresi souboru pomocí zvolených parametrů. Parametry jsou při spouštění zadávány jako argumenty příkazového řádku. Pokud aplikaci spustíme bez parametrů, mohou být následně argumenty zadány pomocí příkazového řádku. Zpracování argumentů obstarává použitá knihovna Mono.Options [19, 20]. Mezi volitelné argumenty konzolové aplikace patří:

- f|file – Specifikuje cestu k souboru, který chceme komprimovat nebo dekomprimovat.
- d|dir|directory – Specifikuje cestu k složce. Všechny soubory v dané složce budou komprimovány nebo dekomprimovány.

- sf|subfolder – Označuje, jestli budou komprimovány nebo dekomprimovány i soubory ve všech podsložkách. Používá se v kombinaci s *d|dir|directory*. Nabývá hodnot *t|true* nebo *f|false*.
- gs|grpblocksize – Udává velikost bloku GRP transformace.
- gm|grpmaxblocksize – Označuje maximální velikost bloku GRP transformace.
- gp|grpparameter – Nastavuje zdroj načítání parametrů GRP transformace. Může nabývat hodnot *file* nebo *memory*.
- gt|grpthreads – Nastavuje maximální počet současně běžících vláken pro GRP transformaci.
- ms|mtfblocksize – Udává velikost bloku MTF transformace.
- mt|mtfthreads – Nastavuje maximální počet současně běžících vláken pro MTF transformaci.
- rs|rleblocksize – Udává velikost bloku RLE algoritmu.
- rt|rlethreads – Nastavuje maximální počet současně běžících vláken pro RLE.
- rdf|rledecodeforce – Nastavuje sekvenční zpracování dekompresní části RLE. Nabývá hodnot *true* nebo *false*.
- at|allthreads - Umožňuje nastavení maximálního počtu všech současně běžících vláken pro všechny metody.
- i|info – Argument, který umožňuje zobrazení dodatečných informací při kompresi a dekompresi.

Některé parametry musí být zadávány tak, aby splňovaly určité podmínky. Pokud jsou parametry zadány nesprávně, je na to uživatel při zadávání upozorněn. Mezi tyto parametry patří například vzájemné nastavení velikosti bloku a velikosti pro generování parametrů. Pokud jsou parametry nastavovány postupně, je každé nastavení potvrzeno výpisem do konzole.

4.4 Průběh implementace a optimalizace kódu

Optimalizace celé knihovny probíhala už od prvotních implementačních fází. Úplně první naimplementovaná verze obsahovala pouze dopřednou a inverzní GRP transformaci. Transformace byla implementována přesně podle návrhu ve zdrojovém článku [1]. Prvotní implementace využívala zcela neefektivní třídící algoritmus Bubblesort, včetně několika slovníkových a listových struktur, které značně zpomalovaly průběh celého výpočtu. Transformace taktéž pracovala s datovým typem char a umožňovala pouze zpracování textových dat. Cílem první implementace však nebyl efektivní návrh. Nejprve jsme museli ověřit, zda daná transformace opravdu

funguje tak, jak byla navržena ve výše zmíněném článku. Prvotní implementace funkčnost potvrdila, současně ale také odhalila několik problémů souvisejících s volbou parametrů. Jak bylo vysvětleno v kapitole číslo 3, jsou parametry voleny na základě určitých podmínek. Z tohoto důvodu bylo nutné navrhnout způsob, jakým se budou parametry pro sestrojení matice vybírat. K první verzi byl proto navržen jednoduchý algoritmus, který generoval všechny možné kombinace vstupních parametrů do určité velikosti vstupu. Parametry byly poté voleny na základě první nalezené kombinace s odpovídající délkou vstupu. V této fázi byla první verze schopna provést dopřednou a inverzní GRP transformaci nad textovými daty. S postupnou optimalizací byly do knihovny přidány i další transformační a kompresní algoritmy.

První spustitelná verze měla do efektivní implementace poměrně daleko. Z tohoto důvodu byly v následujících verzích provedeny změny, které měly pozitivní dopad na výkonnost použitých algoritmů. Mezi tyto změny patří:

1. Optimalizace jednotlivých částí GRP transformace – Oproti původnímu návrhu ve zdrojovém článku byly některé kroky mírně pozměněny. Jednalo se například o spojení třetího a čtvrtého kroku dopředné transformace. Tato část neměla příliš významný podíl na zrychlení, nicméně dokázala značně zpřehlednit zdrojový kód vyřazením nadbytečných kroků transformace.

2. Změna datového typu GRP transformace – V prvotní implementaci byly vykonávány všechny operace s datovým typem char, přičemž entropické kódování pracovalo výhradně s datovým typem Byte. Abychom zabránili pozdějšímu přetypování z charu na Byte, byla transformace přepracována tak, aby pracovala s datovým typem Byte. Tato změna neměla žádný vliv na výkonnost transformace, nicméně odstranila nutnost pozdějšího přetypování.

3. Náhrada neefektivních datových struktur – Jednalo se především o vyřazení velké většiny listů a slovníků, které ve spojení s cyklem definovaným pomoci *foreach* zbytečně zpomalovaly prováděné operace. Pokud jsme neznali velikost, s jakou má být pole vytvořeno, bylo téměř vždy použití listu efektivnější. Pokud jsme velikost znali dopředu, byl list nahrazen polem. V některých případech bylo také možné nahradit slovník pomocí hashsetu (například u RLE).

4. Nahrazení třídících algoritmů GRP transformace – Kvůli své jednoduchosti byl Bubblesort zvolen jako prvotní třídící algoritmus. Bubblesort byl později nahrazen velice výkonným třídícím algoritmem Radixsort. V druhém kroku inverzní transformace byl použit .NET Introsort. Tento krok měl výrazný dopad na zvýšení výkonnosti algoritmu.

5. Cachování parametrů GRP transformace – V kapitole číslo 4.2.2 byl vysvětlen princip ukládání tabulky parametrů. Parametry mohou být uloženy v paměti a není je nutné při kompresi každého souborů znovu vytvářet, což opět přispělo k mírnému zlepšení při kompresi více jak jednoho souboru.

6. Použití serverového garbage collectoru – Běžné pracovní stanice využívají výchozí nastavení GC. Výchozí nastavení vyhledává a uvolňuje paměť současně za běhu klientské aplikace. Použití tohoto typu GC minimalizuje odezvu a zajišťuje stabilní výkonnost aplikace. Naopak serverový GC vytváří pro každé jádro procesoru vlastní haldu a speciální vlákno, které vyhledává a uvolňuje paměť paralelně. Během vyhledávání volné paměti jsou však všechna klientsky vytvořená vlákna pozastavena. Výběr serverového GC značně zvyšuje surový výkon aplikace, nicméně může způsobit drobné prodlevy, které jsou způsobeny pozastavením klientsky vytvořených vláken. V naší aplikaci je zcela jistě preferován surový výkon před stabilní dobou odezvy na uživatelské požadavky, protože uživatel obvykle zadává požadavky pouze na začátku nebo po dokončení zvoleného kompresního procesu. [21]

7. Rozdělení GRP transformace do bloků – I přes všechny úpravy představuje GRP transformace stále výpočetně nejnáročnější část celého kompresního procesu. Z tohoto důvodu byla aplikována podobná strategie, kterou využívají i ostatní kompresní programy. Vstupní data jsou tak rozdělována do jednotlivých bloků, kde každý blok zpracovává jedno vlákno (kapitola číslo 4.2.2).

Výše uvedené změny měly nejrazantnější vliv na efektivitu kompresní knihovny nebo byly z pohledu implementace časově nejnáročnější. Mezi nejznatelnější změny patří implementace Radixsortu, paralelní zpracování a použití serverového garbage collectoru. V celé knihovně byly prováděny i další změny, které určitou částí ovlivnily celkovou výkonnost. Ty však více souvisí se standardním vývojovým procesem než se samotnou optimalizací knihovny.

5 Testování

Testování představovalo nezbytnou část, která doprovázela implementační proces. Testovány byly jednotlivé algoritmy stejně jako celá kompresní aplikace. Všechny testy probíhaly na souborech, které jsou uvedeny v příloze B. V rámci testování byly nejdříve změřeny a porovnány časy výpočtů jednotlivých algoritmů. Poté následovaly testy hodnotící kompresní poměr a v neposlední řadě byly provedeny doplňující testy, které zohledňovaly rozdílné nastavení parametrů. V závěru této kapitoly byly všechny testy vyhodnoceny a porovnány s jinými kompresními algoritmy.

5.1 Testovací korpusy a způsob porovnání

Pro účely testování jsme použili hned několik testovacích korpusů. Testovací korpusy obsahují sadu souborů, které poskytují dostatečné množství testovacích dat. Jednotlivé soubory v použitých korpusech reprezentují jak reálná, tak i uměle vytvořená data. Testovací korpusy záměrně obsahují rozdílná data (text, binární soubory, DNA sekvence, a další), na kterých může být daný kompresní algoritmus efektivně otestován. Testovácí korpusy jsou taktéž vhodné, protože poskytují jednotnou možnost porovnání s ostatními kompresními algoritmy. Pro účely testování byly použity Calgary Corpus [22], Canterbury Corpus [23] a Silesia Corpus [24]. Dále byly použity Artificial Corpus, Large Corpus a Miscellaneous Corpus dostupné z webových stránek Canterbury Corpusu [25]. Mimo testované korpusy byly dále z hlediska doby výpočtu testovány i náhodně generované soubory s postupně narůstající velikostí. Porovnání s ostatními algoritmy a kompresními programy bylo provedeno na základě změřených výsledků a údajů z patřičných kompresních korpusů. Další údaje byly získány z Squash benchmarku [26].

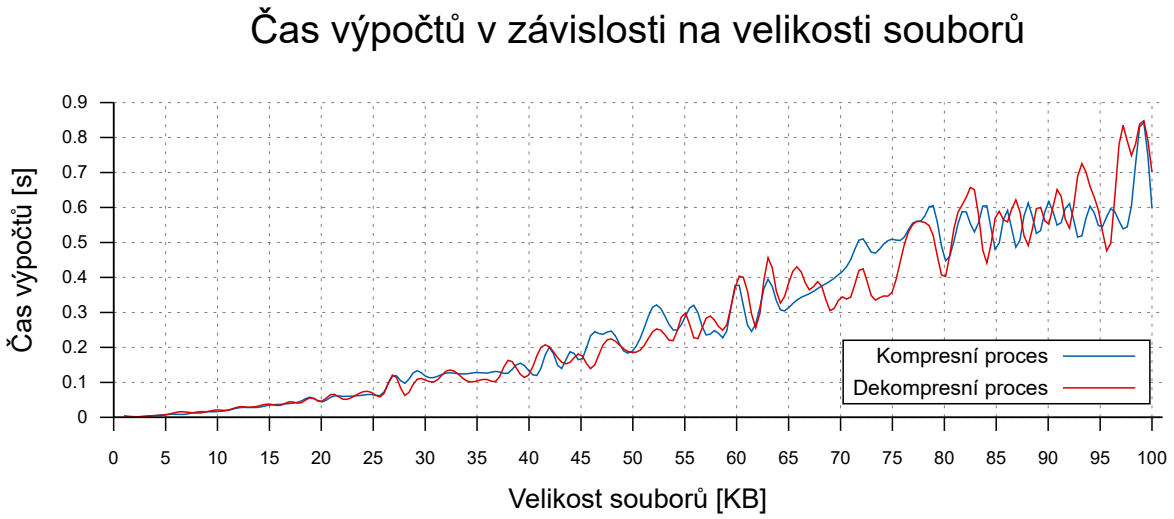
5.2 Čas výpočtů

Čas výpočtů jednotlivých kompresních metod představuje důležitou veličinu. Z tohoto důvodu byly první testy soustředěny na jednotlivé časy výpočtů kompresního a dekompresního procesu. I přesto, že se mohou časy výpočtů na jednotlivých zařízeních lišit, poskytuje tento typ porovnání alespoň obecný pohled na rychlost algoritmu.

5.2.1 Čas výpočtů v závislosti na velikosti souborů

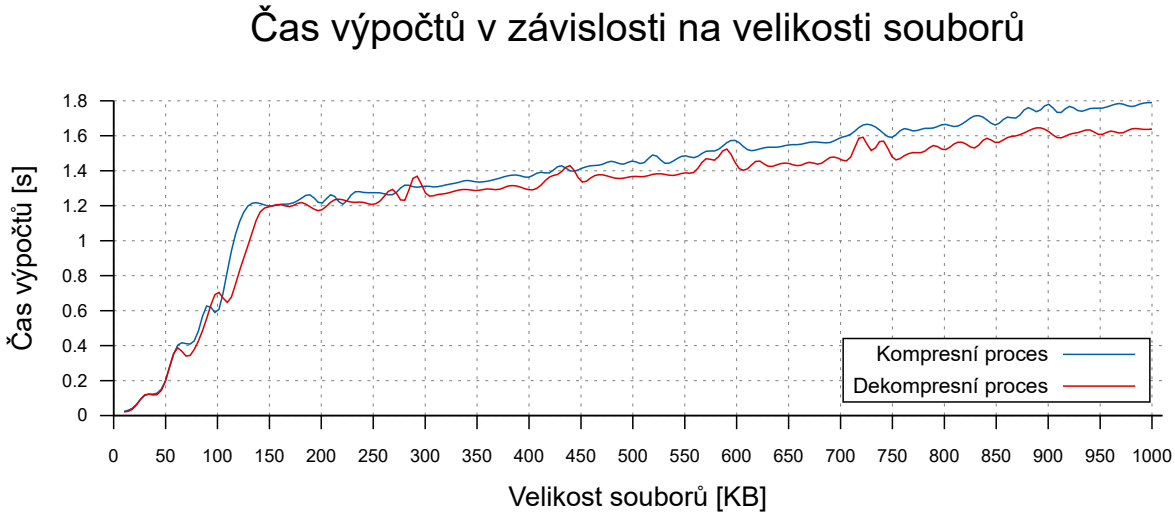
Abychom jednoduše identifikovali, s jakou rychlostí roste čas výpočtů při zvyšující se velikosti vstupu, byly pro tento test vytvořeny 3 sady náhodně generovaných souborů, kde každá sada obsahuje přesně 100 souborů. První z nich obsahuje soubory s velikostí, která se pohybuje v rozmezí 1-100 KiloBytu. Druhá sada obsahuje soubory s velikostí v rozmezí 10-1000 KiloBytu. Třetí sada pak obsahuje soubory s velikostí v rozmezí 100-10000 KiloBytu. Jednotlivé sady byly následně testovány, přičemž velikost GRP bloku měla hodnotu 150 KB. Testování bylo prováděno na celém kompresním procesu.

V první testované sadě můžeme vidět radikální nárůst v době samotného výpočtu. To je zapříčiněno zvyšující se velikostí matice v GRP transformaci. Mírné kolísání v rychlosti výpočtů je taktéž zapříčiněno automaticky volenými parametry pro GRP transformaci. Průběh je zobrazen na obrázků 2.



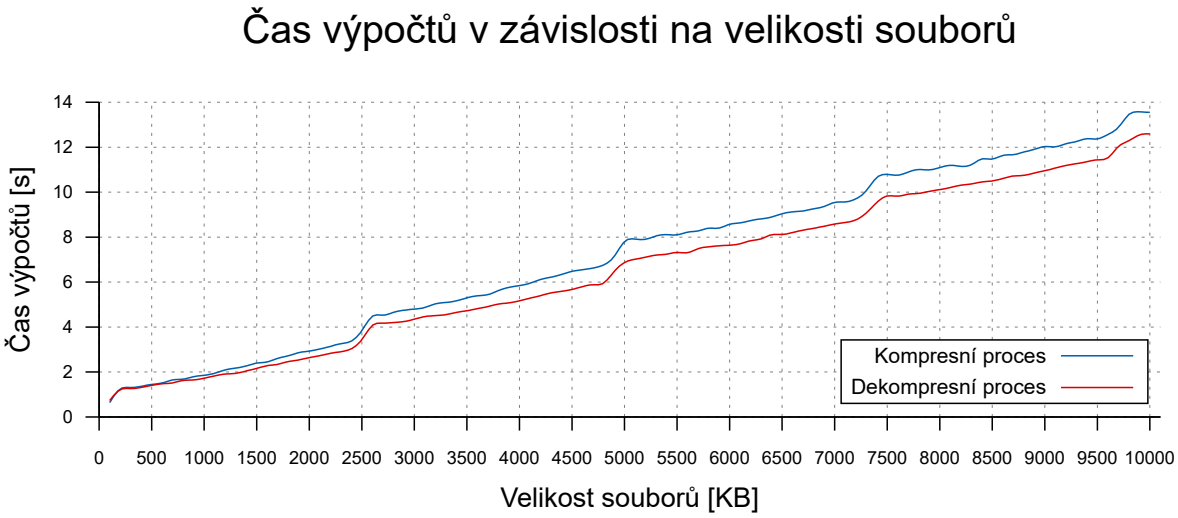
Obrázek 2: Graf - výpočetní čas na malých souborech

V první testované sadě měl největší soubor velikost 100 KB. Vzhledem k tomu, že byla velikost bloku GRP transformace větší než největší soubor z této sady, byly všechny soubory zpracovávány sekvenčně. V druhé testované sadě pokračoval prudký nárůst výpočetního času do přibližně 150 KB (velikost GRP bloku). Od této velikosti lze vidět už jen pozvolný nárůst výpočetního času. Toto je zapříčiněno paralelním zpracováním jednotlivých bloků. Průběh druhé testované sady je zobrazen na obrázku 3.



Obrázek 3: Graf - výpočetní čas na středně velkých souborech

Třetí testovaná sada ukazuje opětovný nárůst ve výpočetním času kompresního a dekompresního procesu. Nárůst je však už poměrně stabilní. Největší soubor o velikost 10 MB byl tak komprimován přibližně 13 sekund, zpětná dekomprese byla o něco málo rychlejší. Průběh poslední testované sady je zobrazen na obrázku číslo 4.



Obrázek 4: Graf - výpočetní čas na velkých souborech

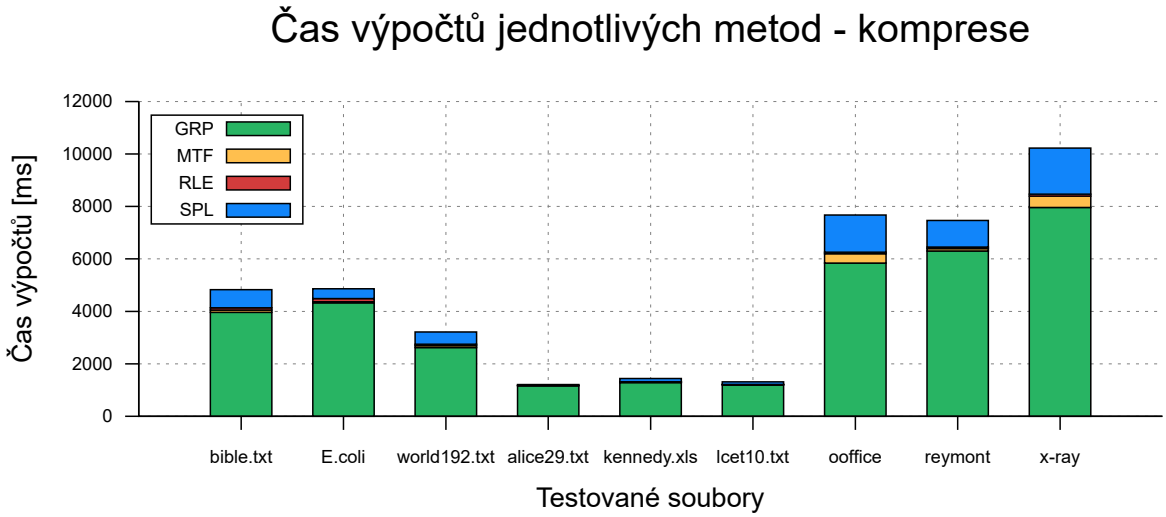
Tyto výsledky představují obrovské zlepšení oproti úplně první implementované verzi, ve které jsme komprimovali soubor alice29.txt o velikosti 152 KB přibližně 72 sekund. V této verzi jsme obdobně velký soubor komprimovali přibližně $1,2$ sekundy. Z prezentovaných výsledků lze odvodit poměrně velkou výpočetní náročnost celého kompresního a dekompresního procesu. Z tohoto důvodu budou u dalších testů prezentovány i časy jednotlivých kompresních metod. Zaznamenávání kompresního poměru je u náhodně generovaných souborů zbytečné, a proto zde byly uvedeny pouze jednotlivé časy výpočtů.

5.2.2 Čas výpočtů u testovacích souborů

V předchozí části jsme testovali pouze náhodně generované soubory, u kterých jsme zaznamenávali výpočetní čas komprese a dekomprese. V této části jsme otestovali veškeré soubory z kompresních korpusů. U každého souboru byl zaznamenán výpočetní čas pro každou z použitých kompresních metod. Kompletní výsledky tohoto testu jsou zobrazeny v tabulkách číslo 16 a 17.

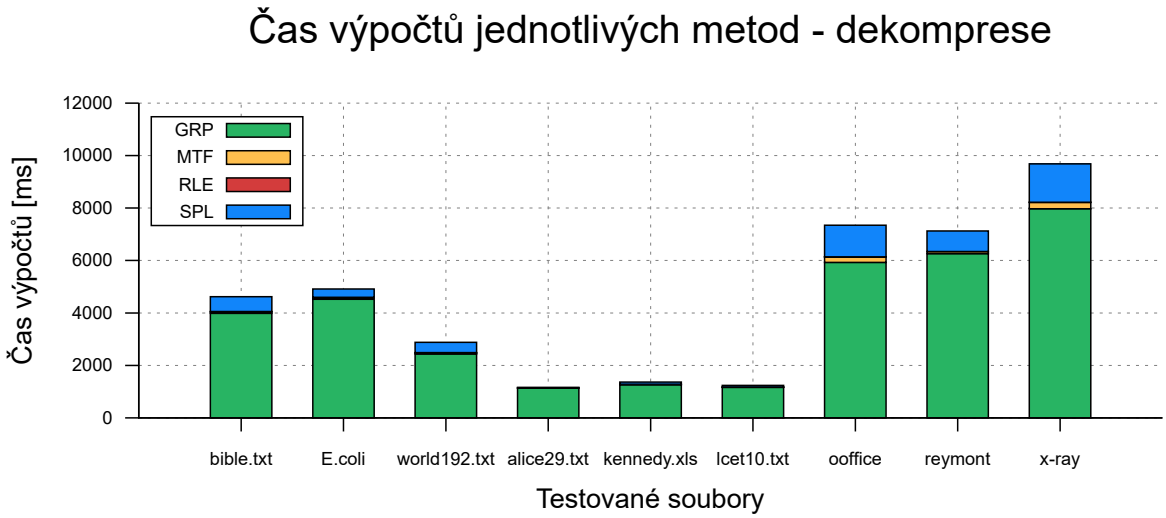
Na obrázku číslo 5 vidíme graficky znázorněný výběr z několika testovaných souborů. U těchto souborů můžeme pozorovat hodnoty, které přibližně odpovídají stejně velkým souborům v předešlém testu. Na rozdíl od prvního testu zde vidíme i výpočetní časy jednotlivých metod. Jak už bylo naznačeno, největší část výpočtů reprezentuje GRP transformace, a to i přes veškerou optimalizaci, kterou tato transformace prošla. Přibližně 80 % celkového výpočetního času zabírá GRP transformace. Komprese pomocí Splay Tree zabírá přibližně dalších 15 %. Zby-

lých 5 % je pak rozloženo mezi MTF transformací a RLE. Zde platí, že je MTF transformace obvykle o něco náročnější než RLE, a to zvláště u větších souborů.



Obrázek 5: Graf - výpočetní čas jednotlivých metod - komprese

Na obrázku číslo 6 vidíme graficky znázorněný dekompresní proces, který byl spuštěn na stejné sadě souborů. Z grafu vyplývá, že je dekompresní proces o něco málo rychlejší než proces kompresní. To taktéž odpovídá zjištění v testu, který byl uveden v kapitole číslo 5.2.1. Na obou grafech lze taktéž vypozorovat, kdy byl spuštěn paralelní výpočet. Soubor alice29.txt s velikostí 152 KB byl zpracováván téměř stejně dlouhou dobu jako soubor kennedy.xls, který má velikost přibližně 1 MB. Velikost bloku byla stejně jako v předchozím případě nastavena na 150 KB.



Obrázek 6: Graf - výpočetní čas jednotlivých metod - dekomprese

5.3 Kompresní poměr

Kompresní poměr poskytuje spolu s výpočetním časem důležitý ukazatel celkové výkonosti konkrétního kompresního algoritmu. V této části se zaměříme na vyhodnocení kompresního poměru, kompresního faktoru a dalších metod porovnání. Testována byla základní verze včetně dalších potencionálně zajímavých kombinací kompresních algoritmů.

5.3.1 Výsledky základní varianty

Základní varianta reprezentuje návrh, který byl představen v kapitole číslo 4.1. Kompletní výsledky této testované varianty jsou zobrazeny v tabulkách 18 a 19. Průměrné hodnoty na všech testovaných korpusech jsou pak zobrazeny v tabulce číslo 4.

Tabulka 4: Průměrné hodnoty metod porovnání

Testovaný korpus	$\overline{x}(\alpha)$	$\overline{x}(f)$	$\overline{x}(s)$	$\overline{x}(bps)$
Canterbury	0,645	1,912	0,355	5,156
Artificial	0,319	95,783	0,681	2,550
Large	0,551	2,045	0,449	4,413
Miscellaneous	0,498	2,008	0,502	3,985
Calgary	0,729	1,563	0,271	5,832
Silesia	0,684	1,642	0,316	5,472

Z přiložených výsledků vidíme, že navržená kompresní metoda opravdu funguje. Nejlépe dopadly testované Artificial a Miscellaneous korpusy. Artifical korpus obsahuje soubory, které jsou generované a jednoduše komprimovatelné. Miscellaneous korpus obsahuje pouze jeden soubor se zapsanou hodnotou Π (prvních 100 000 číslic). U tohoto souboru jsme schopni reprezentovat každý znak 4 bity jednoduchou úpravou. Vidíme tak, že ostatní metody (mimo binární kódování) celkovému kompresnímu poměru moc nepomohly.

V Canterbury korpusu byly nejlépe komprimované soubory *kennedy.xls* a *ptt5*. Na druhou stranu, komprese textových souborů poměrně zaostávala. Průměrná hodnota bps byla $5,156$, přičemž jsme průměrně ušetřili přibližně 35,5 % potřebného místa na jeden soubor. U ostatních korpusů jsme dosahovali velice podobných výsledků.

5.3.2 Výsledky experimentálních variant

Pro účely komprese byly vytvořeny další experimentální varianty s použitím různých kombinací jednotlivých algoritmů. Tyto experimentální varianty byly porovnány se základní variantou a dále vyhodnoceny. Pro tento test byly vybrány soubory z různých korpusů s velikostí GRP bloku 200 KB. Mezi testované varianty patří:

- V1 – Základní definovaná verze (kapitola číslo 4.1).

- EV1 – U této varianty používáme binární zápis v RLE. Varianta nahrazuje posloupnosti nul a jedniček patřičnými bity.
- EV2 – V této variantě využíváme stejné změny jako v EV1, dále je zde Splay Tree kódování nahrazeno Huffmanovým kódováním.
- EV3 – Varianta, která používá binární zápis v RLE. Tato varianta zapisuje 2 bity na výstup v případě, kdy hodnota Bytu z MTF výstupu nabývá hodnot 0-3.
- EV4 – V této variantě využíváme stejné změny jako v EV3, dále je zde Splay Tree kódování nahrazeno Huffmanovým kódováním.
- EV5 – Varianta, která využívá Huffmanovo kódování místo použitého Splay Tree kódování.
- EV6 – Tato varianta naprosto vypouští posloupnost MTF transformace a RLE. Provedena je tedy pouze GRP transformace a následné Splay Tree kódování.
- EV7 – Tato varianta taktéž vypouští posloupnost MTF a RLE. Nejdříve je provedena GRP transformace a následně Huffmanovo kódování.

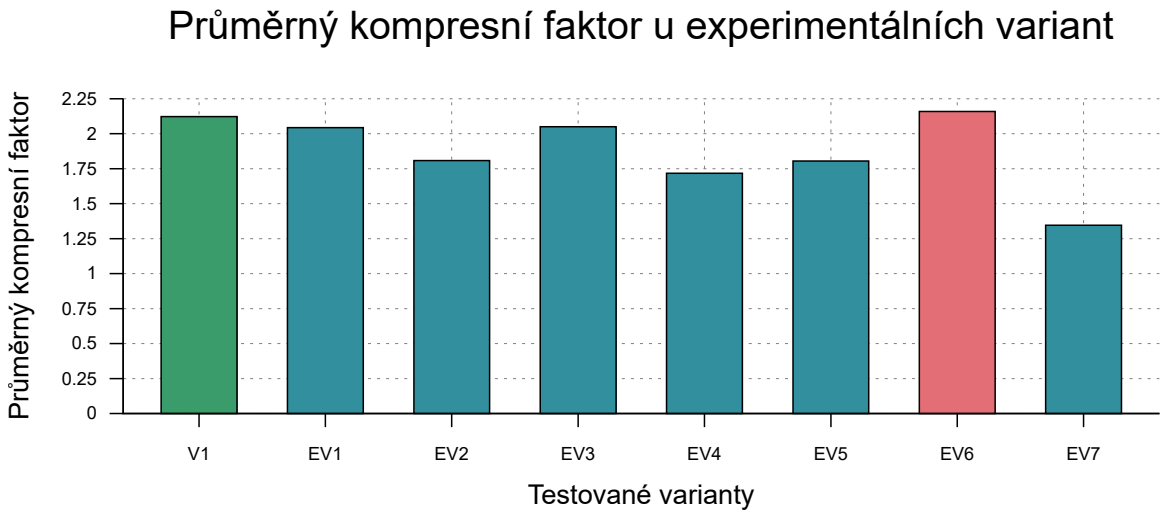
Výsledky těchto variant jsou zobrazeny v tabulce číslo 5. Všechny hodnoty jsou zaznamenány pomocí kompresního faktoru. Pokud patřičná varianta využívala Huffmanovo kódování, byl celkový kompresní faktor počítán z velikosti, která obsahovala uložené frekvence pro zpětnou dekompresi.

Tabulka 5: Výsledky experimentálních variant

Název souboru	Kompresní varianty [f]							
	V1	EV1	EV2	EV3	EV4	EV5	EV6	EV7
alice29.txt	1,492	1,449	1,715	1,427	1,692	**1,738**	1,673	1,724
asyoulik.txt	1,428	1,399	1,518	1,379	1,501	1,528	**1,594**	1,500
bible.txt	1,607	1,596	1,143	1,558	1,088	1,085	**1,853**	1,012
E.coli	3,124	2,644	2,463	3,316	1,908	2,652	**3,350**	2,666
geo	1,404	1,414	1,071	1,378	1,295	0,969	**1,490**	0,666
kennedy.xls	3,000	**3,201**	2,364	3,089	2,447	1,639	2,899	1,070
lcet10.txt	1,461	1,453	1,342	1,430	1,361	1,382	**1,663**	1,225
news	1,247	1,238	1,169	1,232	1,189	1,160	**1,389**	1,123
plrabn12.txt	1,510	1,499	1,430	1,467	1,399	1,427	**1,715**	1,268
ptt5	**5,666**	5,181	4,332	4,878	3,665	4,941	4,553	1,339
world192.txt	1,404	1,409	1,342	1,395	1,351	1,337	**1,570**	1,216
$\overline{x}$	2,122	2,044	1,808	2,050	1,718	1,805	**2,159**	1,346

Z přiložených výsledků vidíme jasnou převahu šesté experimentální varianty. Tato varianta využívá pouze GRP transformaci s následným Splay Tree kódováním. I přes adaptivní chování Splay Tree kódování je tento výsledek docela překvapivý. Z tohoto výsledku vyplývá fakt, že MTF

a RLE kroky jsou při použití Splay Tree kódování spíše na škodu. Tyto dvě metody upravují data tak, že výstupní data z RLE jsou hůře komprimovatelná než původní výstupní data z GRP transformace. Na druhou stanu Huffmanovo kódování z provedené MTF transformace a RLE profitovalo. Tento výsledek však nebyl dostatečný pro to, aby překonal výsledky Splay Tree komprese. Pro úplnou přehlednost byly průměrné hodnoty graficky znázorněny na obrázku číslo 7. Zelenou barvou je označena původní navržená varianta, modrá barva označuje experimentální varianty, červeně označený sloupec reprezentuje nejlepší variantu.



Obrázek 7: Graf - průměrný kompresní faktor u testovaných experimentálních variant

5.4 Vliv nastavení na GRP transformaci

V této části byly testovány parametry samotné GRP transformace. Mezi tyto parametry patří kontextové uspořádání (d) a velikost bloku (l). Pro tyto parametry byly v kapitole 3.1 definovány určité podmínky, které musí být splněny. V této části testování je pro nás důležitá podmínka $d < l$. Testovány byly soubory uvedené v tabulce číslo 5. Všechny parametry byly voleny automaticky. Pro konkrétní parametry však byly stanoveny další doplňující podmínky, které omezily možný počet všech kombinací. Varianty jsou uvedeny v tabulce číslo 6.

Tabulka 6: Definované varianty parametrů

Varianta	podmínka	příklad, kde $l = 300$
EP1	$d > l - l/1$	$0 < d < 300$
EP2	$d > l - l/2$	$150 < d < 300$
EP3	$d > l - l/3$	$200 < d < 300$
EP4	$d > l - l/6$	$250 < d < 300$
EP5	$d > l - l/20$	$285 < d < 300$
EP6	$d < l - l/3$	$0 < d < 200$

Vliv parametrů na kompresní poměr byl potvrzen, nicméně změny v kompresním poměru byly při rozdílných parametrech spíše minimální (po zaokrouhlení jsou některé hodnoty stejné). Obecně platí, že menší hodnota parametru d negativně ovlivňuje kompresi. Pokud byl parametr d omezen podmínkou zdola, byl kompresní faktor obvykle neměnný. To je zapříčiněno automatickou volbou parametrů, která preferuje vyšší hodnotu parametru d. Varianty EP1 – EP5 proto dosahují obvykle stejných nebo velice podobných hodnot. U varianty EP6 byl parametr d omezen shora. V tomto případě vidíme u téměř všech souborů zhoršení ve výsledném kompresním faktoru. Výjimkou je soubor *ptt5*. U něj je detekováno zlepšení u poslední testované varianty. To je pravděpodobně zapříčiněno tím, že tento soubor obsahuje velmi časté posloupnosti stejných Bytu. Vyšší hodnota parametru d demonstruje větší množství provedených operací nad těmito daty. Následné úpravy pak tyto posloupnosti stejných Bytu spíše znehodnocují, což vyústilo v lepší kompresní faktor u menší hodnoty d. Výsledky jsou zobrazeny v tabulce číslo 7.

Tabulka 7: Výsledky doplňujících podmínek pro parametr d

Název souboru	Kompresní varianty parametrů $[f]$					
	EP1	EP2	EP3	EP4	EP5	EP6
alice29.txt	1,483	1,495	1,495	**1,499**	1,498	1,483
asyoulik.txt	1,428	1,428	1,428	**1,428**	1,425	1,398
bible.txt	1,638	1,638	1,638	**1,638**	1,638	1,623
E.coli	3,124	3,124	3,124	3,124	**3,124**	3,123
geo	**1,404**	1,404	1,178	1,402	1,178	1,404
kennedy.xls	**2,944**	2,926	2,925	2,925	2,923	2,935
lcet10.txt	1,482	1,484	1,484	1,484	**1,485**	1,470
news	1,258	1,258	1,258	**1,262**	1,261	1,255
plrabn12.txt	1,531	1,531	1,531	**1,532**	1,531	1,521
ptt5	5,549	5,549	5,549	5,549	5,486	**5,512**
world192.txt	1,430	1,430	1,430	1,430	**1,430**	1,417

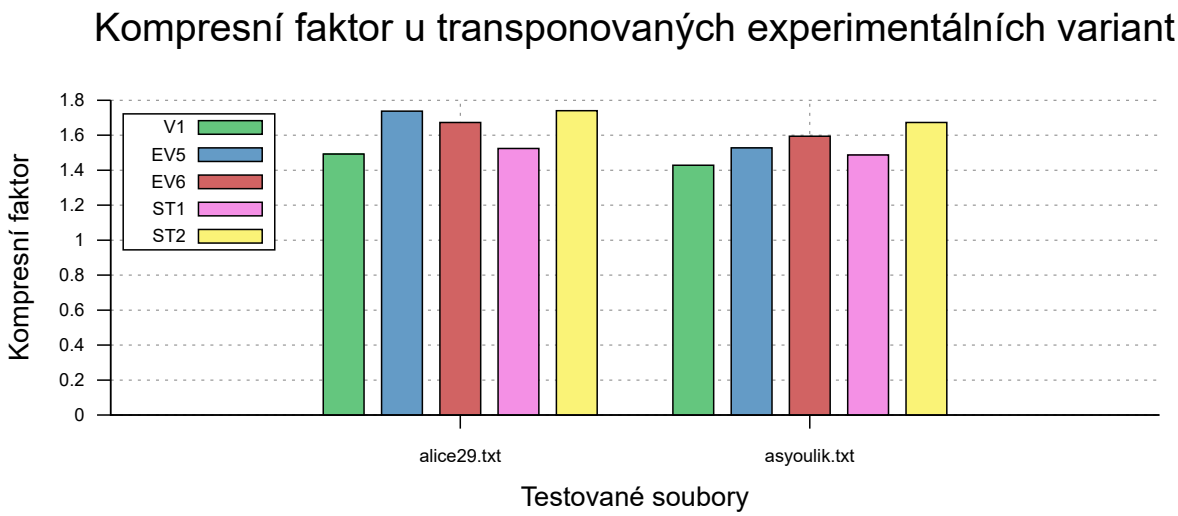
5.5 Řádkový výstup a transponovaná matice výstupu

Doposud byl výstup transformace chápán jako jednotný řetězec, který byl dále komprimován. GRP transformace však produkuje výstup v jednotlivých řádcích matice (dopředná transformace – krok číslo 3). Podle původního návrhu byly řádky spojeny a výstup představuje pouze jeden řetězec. Ve skutečnosti jsme však schopni lépe komprimovat tyto řádky než celý výstup najednou. Komprese jednotlivých výstupních řádků by mohla být prováděna pomocí slovníkových metod, které by nahrazovaly předchozí výskyty podřetězců v jednotlivých řádcích. Stejně tak by mohlo být dále vytvořeno entropické kódování, které využívá malé velikosti abecedy v jednotlivých řádcích výstupu. Toto rozšíření však nebude v této práci implementováno a slouží spíše jako návrh pro budoucí rozšíření.

Při postupném testování byla objevena další vlastnost, kterou můžeme využít k výslednému zlepšení kompresního poměru. Oproti BWT vytváří tato transformace úseky (výše zmíněné řádky výstupu), které disponují nízkou velikostí abecedy. V GRP transformaci jsou stejné znaky obvykle shlukovány na začátku těchto jednotlivých řádků. Konec těchto řádků je daleko více rozmanitý než začátek, což je pravděpodobně ovlivněno parametrem d. Pokud bychom tedy byli schopni transformaci modifikovat do takové podoby, ve které můžeme nastavit libovolně velký parametr d, pak bychom pravděpodobně dosáhli vyššího kompresního poměru. Bez samotného ověření však tento předpoklad potvrdit nemůžeme.

Současně navržené řešení však nevyužívá vlastnosti, kdy jsou na začátků jednotlivých řádků výstupu shlukovány stejné Byty nebo symboly napříč všemi řádky. Z tohoto důvodu byl v této části otestován a navržen čtvrtý krok dopředné GRP transformace. Čtvrtý krok navazuje na třetí krok dopředné transformace a vytváří matici $B = (a_{ij})_{l \times b}$. Do této matice vkládáme postupně po řádcích všechny Byty nebo znaky z výstupního řetězce. K této matici vytvoříme transponovanou matici B^T. Z matice B^T pak následně (po řádcích) získáváme výstupní řetězec. Vytvoření nadbytečné matice slouží pouze pro názornost příkladu.

Experimentální čtvrtý krok byl taktéž otestován na dvou souborech *alice29.txt* a *asyoulik.txt*, které jsou součástí Canterbury korpusu. Porovnání bylo provedeno s výsledky v kapitole 5.3.2. Znázorněné výsledky vidíme na obrázku číslo 8. Konkrétní výsledky jsou pak zaznamenány v tabulce číslo 20 a 21.



Obrázek 8: Graf - kompresní faktor transponovaných experimentálních variant

Na přiloženém grafu vidíme původní navrženou variantu V1, která je označená zelenou barvou. Modře je označena varianta EV5, která dosáhla u předchozího testu nejlepšího výsledku na souboru *alice29.txt*. Červenou barvou je označena varianta EV6. Ta dosáhla nejlepšího výsledku v předchozím testu u souboru *asyoulik.txt*. První nově definovaná varianta je ST1 (označená růžovou barvou), která používá GRP transformaci s transponováním, MTF transfromaci, RLE

a Splay tree kompresi. Druhou nově definovanou variantou je ST2. Ta používá pouze GRP transformaci s transponováním a následnou Splay tree kompresi. Tato varianta je označena žlutou barvou.

Z výsledků je patrné, že varianta ST2 dosáhla na nejvyšší kompresní faktor v obou případech. U souboru *alice29.txt* vidíme mírné zlepšení oproti průměrné nejlepší variantě EV6. Rozdíl mezi nejlepší variantou pro tento konkrétní soubor a variantou ST2 je zanedbatelný. I přesto je však varianta ST2 na prvním místě. U druhého souboru *asyoulik.txt* vidíme další zlepšení o přibližně 5 %. Zlepšení je uvedeno v závislosti na nejlepší variantě (EV6) pro tento konkrétní soubor.

5.6 Porovnání výsledků a vyhodnocení

V této části jsou stručně shrnuty dosažené výsledky, které jsou následně porovnány s ostatními kompresními metodami.

5.6.1 Porovnání výsledků u Canterbury korpusu

V tabulce číslo 8 vidíme uvedené výsledky dvou variant této experimentální aplikace. Spolu s těmito výsledky jsou zde uvedeny i obdobné výsledky ostatních kompresních metod. Uvedené hodnoty jsou zapsány v bps. Původní navržená varianta měla průměrnou hodnotu bps 5,14. Z dalších testovaných variant dopadla nejlépe varianta ST2, která měla průměrnou hodnotu bps 4,29. Celkově se však tato metoda umístila v dolní části tabulky.

Tabulka 8: Porovnání s ostatními kompresními programy - Canterbury korpus

#	Kompresní metoda	$\overline{x}(bps)$
1	ppmD5	2, 11
2	ppmD7	2, 15
3	bzip-6	2, 15
⋮	⋮	⋮
25	lzrw1	4, 18
26	huffword2	4, 20
27	**grp-st2**	**4,29**
28	char	4, 49
29	pack	4, 53
30	**grp-v1**	**5,14**
31	yabba512	5, 19
32	cat	8, 00

I přes nepříliš přesvědčivý výsledek lze vidět znatelné zlepšení mezi původní navrženou variantou `grp-v1` a variantou `grp-st2`, která využívá transponovanou matici. Rozdíl mezi těmito variantami je 0,85 bps, což představuje zlepšení o přibližně 17,5 %.

5.6.2 Porovnání výsledků pomocí Squash benchmarku

Squash benchmark poskytuje v současné době výsledky 278 kompresních metod, což zahrnuje i různé implementace a varianty jednotlivých kompresních metod. Bohužel zde nejsou uvedeny výsledky pro všechny soubory z jednotlivých testovacích korpusů, a proto byly porovnány jen vybrané soubory z testovaných korpusů. Všechny hodnoty jsou zaznamenány v jednotkách kompresního faktoru.

V tabulce číslo 9 vidíme porovnávaný soubor *plrabn12.txt*. Tento soubor dokázal nejlépe komprimovat zpaq, který využívá Content-mixing. Zpaq taktéž obsahuje hned několik dalších metod, jako jsou například LZ77 nebo BWT. Tyto metody jsou použity pro cílenou kompresi na základě charakteru vstupních dat. Na čtvrtém místě se umístil bzip2, který využívá BWT transformaci. Naše experimentální aplikace se umístila na 209. místě. U již několikrát zmiňovaného souboru *alice29.txt* se naše aplikace umístila na 242. místě (22). Průměrně se pak tato metoda umístila v poslední čtvrtině všech testovaných kompresních metod.

Tabulka 9: Squash benchmark - soubor *plrabn12.txt*

#	Kompresní metoda	Verze	Úroveň	f
1	zpaq	zpaq	5	3,77
2	zpaq	zpaq	4	3,60
3	bsc	bsc	-	3,54
4	bzip2	bzip2	5	3,31
⋮	⋮	⋮	⋮	⋮
208	lzo	lzo1c	99	1,83
209	**grp**	**grp-st2**	**-**	**1,81**
210	lzo	lzo1b	8	1,81
⋮	⋮	⋮	⋮	⋮
278	lz4	lz4	1	1,03

5.6.3 Finální vyhodnocení

Dnešní kompresní programy obvykle kombinují celou řadu různých kompresních metod, které jsou voleny na základě rozdílného typu vstupních dat. Pro příklad můžeme uvést více než 20 let starý bzip, který využívá hned několik Huffmanových tabulek, mezi kterými vybírá tu nejlepší právě na základě typu komprimovaných dat. Těmto komplexním a implementačně rozsáhlým kompresním programům se ve výsledku tato experimentální aplikace nemůže vyrovnat. To však nebylo ani cílem této práce. Tato práce měla efektivně zhodnotit teoreticky navrženou GRP transformaci v praktické kompresi dat. GRP transformace v této podobě nepředstavuje nejefektivnější algoritmus, který napomáhá výsledné kompresi. Na druhou stranu však důkladné testování odhalilo nespočet různých možností a celkově zhodnotilo nedostatky současného návrhu. Ty mohou být budoucí úpravou odstraněny, což poskytuje prostor pro další experimenty.

6 Závěr

Cílem této diplomové práce bylo nastudování a implementace experimentální aplikace, která využívá zobecněnou transformaci dat. Vytvořená aplikace měla být následně otestována nad rozdílnými daty. Souběžným cílem bylo také nastudování a popis jednotlivých kompresních metod, které byly použity v kombinaci se zobecněnou transformací dat. Výsledek této práce představuje implementovaná C# knihovna pro kompresi dat. K samotné knihovně je implementovaná konzolová aplikace, která demonstruje použití kompresní knihovny. Výsledná práce taktéž zahrnuje textovou část, kde jsou stručně objasněny principy ztrátové a bezeztrátové komprese dat. V textové části jsou dále popsány použité algoritmy s uvedenými příklady. V neposlední řadě jsou zde uvedeny popisy implementovaných tříd kompresní knihovny a vyhodnocené testy, které jsou popsány a znázorněny pomocí jednotlivých tabulek a grafů.

Zobecněná transformace dat byla implementována podle teoretického návrhu a dále vylepšena odstraněním některých nadbytečných kroků. K této transformaci byly implementovány další kompresní algoritmy, které tvoří jednotný kompresní proces. Transformace a kompresní algoritmy byly následně přepracovány do takové podoby, která podporuje paralelní zpracování dat. Pro účely testování byly dále vytvořeny rozdílné kompresní varianty. Každá varianta byla odlišena dostatečně znatelnou úpravou nebo výměnou specifického kompresního algoritmu. U výchozí varianty byl následně otestován a změřen výpočetní čas. Rozdílné varianty následně sloužily pro celkové porovnání kompresního poměru. Varianty byly testovány na souborech z testovacích korpusů. Celkové výsledky byly na závěr porovnány s dalšími kompresními programy a algoritmy.

I přes relativně pozitivní výsledky se tato experimentální aplikace nemůže kompresním poměrem srovnávat s dalšími kompresními programy. Podrobné testování však odhalilo určité vlastnosti, kterými transformace disponuje. Tyto vlastnosti mohou být plně využity v dalším rozšíření GRP transformace. Samotná transformace by mohla být dále optimalizována do takové podoby, kde budeme schopni volit parametry, které nemusí splňovat počáteční podmínky. Tím je zvláště myšlena modifikace transformace pro $d \geq l$. Mezi další modifikace by mohlo byt zařazeno kódování jednotlivých řádků výstupu. Ačkoli byl v této práci považován výstup dopředné transformace za jednotný řetězec, je výstup ve skutečnosti reprezentován jednotlivými řádky, které obvykle obsahují vhodná data pro následnou slovníkovou kompresi nebo kompresi pomocí specifického entropického algoritmu s omezenou délkou abecedy. Za zmínku taktéž stojí objevený vztah transponované matice výstupu, který je v některých případech lépe komprimovatelný než původně prezentovaný model výstupu.

Literatura

[1] INAGAKI, K., TOMIZAWA Y., YOKOO H. Novel and Generalized Sort-Based Transform for Lossless Data Compression. *SPIRE*. 2009, s. 102-113, LNCS 5721.

[2] BURROWS, M., WHEELER, D.J. *A block-sorting lossless data compression algorithm.* SRC Research Report 124. Palo Alto: Digital Systems Research Center, 1994.

[3] VO, B.D., MANKU, G.S RadixZip: Linear time compression of token streams. *Very Large Data Bases*. Vienna, 2007, (Proc. 33rd Intern), s 1162–1172.

[4] PU, Ida Mengyi. Fundamental data compression. Burlington, MA: Butterworth-Heinemann, 2006.

[5] SAYOOD, Khalid. *Introduction to data compression.* 2nd ed. San Francisco: Morgan Kaufmann Publishers, c2000. Morgan Kaufmann series in multimedia information and systems. ISBN 1-55860-558-4.

[6] SALOMON, David. *Data Compression, The Complete Refference.* London: Springer, 2006. Fourth edition. ISBN-10 1-84628-602-6.

[7] *github: imgmin* [online]. [cit. 2018-04-13]. Dostupné z: https://github.com/rflynn/imgmin

[8] *Czech graphemes frequencies* [online]. [cit. 2018-04-13].
Dostupné z: http://www.czech-language.cz/alphabet/alph-prehled.html

[9] ZIV J., LEMPEL A. A universal algorithm for sequential data compression.*IEEE Transactions on Information Theory.* 1977, (IT-23). ISSN: 0018-9448.

[10] ZIV J., LEMPEL A. Compression of individual sequences via variable rate coding.*IEEE Transactions on Information Theory.* 1978, (IT-24). ISSN: 0018-9448.

[11] WELCH T. A., LZW - A Technique for High-Performance Data Compression, *Computer*, 1984, (vol. 17), s. 102-113.

[12] ADJEROH D., BELL T., MUKHERJEE A., *The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching.* Heidelberg: Springer, 2008.

[13] *geeksforgeeks: Radixsort* [online]. [cit. 2018-04-14].
Dostupné z: https://www.geeksforgeeks.org/radix-sort/

[14] HUFFMAN D.A., A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE.* 1952, (Volume: 40, Issue: 9), s. 1098 - 1101. DOI: 10.1109/JRPROC.1952.273898. ISSN 0096-8390.

[15] MYŚLIWIEC, K. *Implementation of Huffman Coding algorithm with binary trees* [online]. [cit. 2018-04-14]. Dostupné z: https://kamilmysliwiec.com/implementation-of-huffman-coding-algorithm-with-binary-trees

[16] SLEATOR D.D., TARJAN R.E. Self-adjusting binary search trees. *Journal of the ACM.* 1985, (32), 652–686.

[17] *Data structures - Splay Tree* [online]. [cit. 2018-04-14]. Dostupné z: http://btechsmartclass.com/DS/U5_T5.html

[18] *BZIP* [online]. [cit. 2018-04-14]. Dostupné z: http://www.bzip.org/

[19] *Mono Options Documentation* [online]. [cit. 2018-04-14]. Dostupné z: https://github.com/xamarin/XamarinComponents/tree/master/XPlat/Mono.Options

[20] *Mono Options source* [online]. [cit. 2018-04-14]. Dostupné z: https://github.com/mono/mono/blob/master/mcs/class/Mono.Options/Mono.Options

[21] *C# Server Garbage Collector* [online]. [cit. 2018-04-14]. Dostupné z: https://msdn.microsoft.com/en-us/library/cc165011(v=office.11).aspx

[22] BELL T. C., GLEARY J. G., WITTEN I. H. *Text Compression.* Prentice Hall, Englewood Cliffs, 1990.

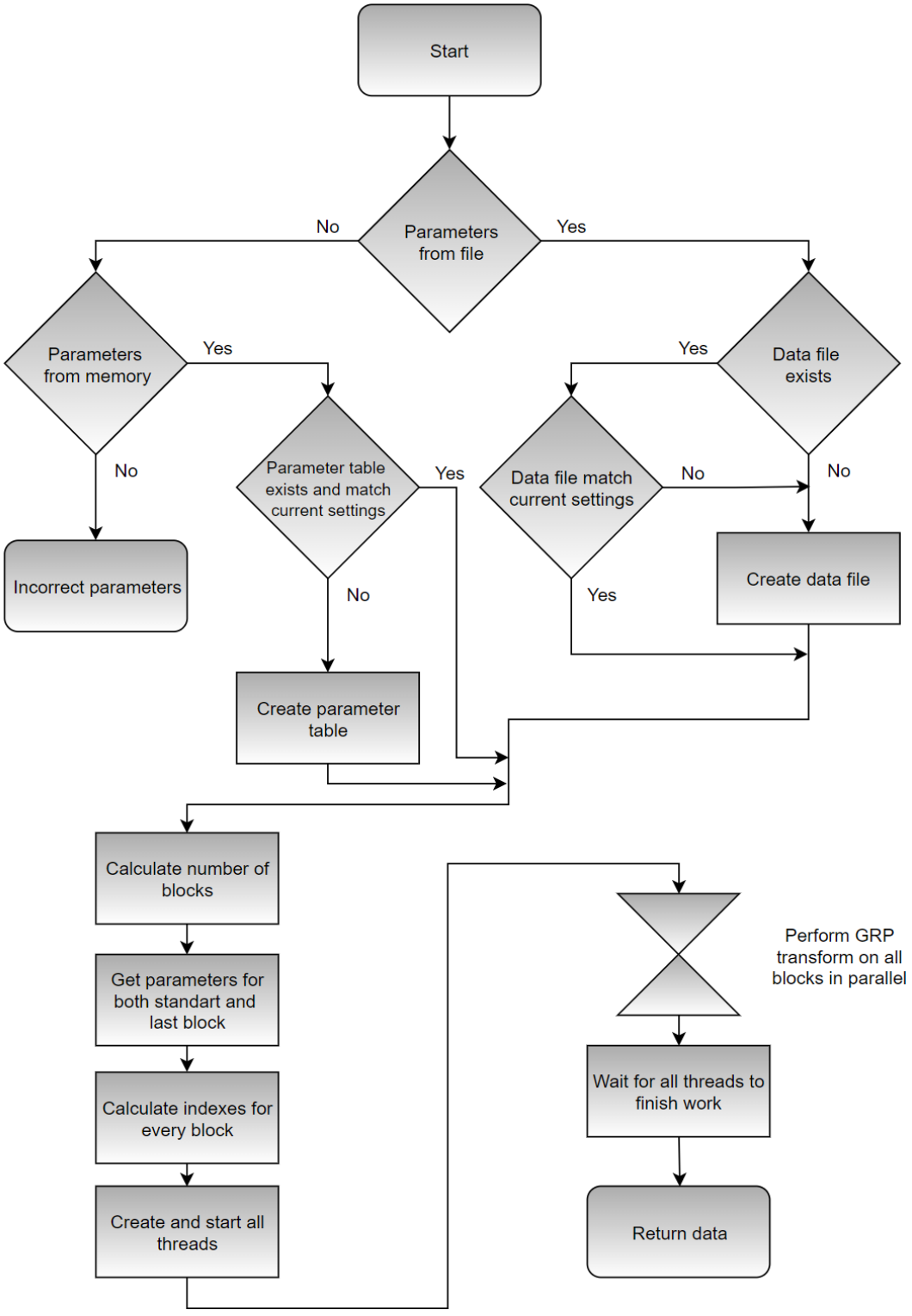
[23] BELL T., ARNOLD R., *A corpus for the evaluation of lossless compression algorithms.* Christchurch, 1997. ISBN 0-8186-7761-9. Department of Computer Science, University of Canterbury.

[24] *Silesia Corpus* [online]. [cit. 2018-04-14]. Dostupné z: http://sun.aei.polsl.pl/ sdeor/index.php?page=silesia

[25] *Artificial Corpus, Large Corpus a Miscellaneous Corpus* [online]. [cit. 2018-04-14]. Dostupné z: http://corpus.canterbury.ac.nz/descriptions/

[26] *Squash Benchmark* [online]. [cit. 2018-04-14]. Dostupné z: https://quixdb.github.io/squash-benchmark/

A Vývojový diagram



Obrázek 9: Vývojový diagram - rozdělení a incializace GRP transformace

B Seznam testovaných souborů

Tabulka 10: Canterbury Corpus

Název souboru	Zkratka	Kategorie	Velikost [Byte]
alice29.txt	text	Anglický text	152 089
asyoulik.txt	play	Shakespeare	125 179
cp.html	html	HTML - zdrojový kód	24 603
fields.c	Csrc	C - zdrojový kód	11 150
grammar.lsp	list	LISP - zdrojový kód	3 721
kennedy.xls	Excl	Excel soubor	1 029 744
lcet10.txt	tech	Technická práce	426 754
plrabn12.txt	poem	Poezie	481 861
ptt5	fax	Testovací sada CCITT	513 216
sum	SPRC	SPARC exe soubor	38 240
xargs.1	man	GNU manuál	4 227

Tabulka 11: Artificial Corpus

Název souboru	Zkratka	Kategorie	Velikost [Byte]
a.txt	a	znak 'a'	1
aaa.txt	aaa	100 000 znaků 'a'	100 000
alphabet.txt	alphabet	opakujících se 26 znaků abecedy	100 000
random.txt	random	náhodné znaky	100 000

Tabulka 12: Large Corpus

Název souboru	Zkratka	Kategorie	Velikost [Byte]
E.coli	E.coli	E. Coli genome	4 638 690
bible.txt	bible	Bible - verze krále Jakuba	4 047 392
world192.txt	world	CIA kniha světových faktů	2 473 400

Tabulka 13: Miscellaneous Corpus

Název souboru	Zkratka	Kategorie	Velikost [Byte]
pi.txt	pi	První milion číslic π	1 000 000

Tabulka 14: Calgary Corpus

Název souboru	Zkratka	Kategorie	Velikost [Byte]
bib	bib	Bibliografie	111 261
book1	book1	Fikce (kniha)	768 771
book2	book2	Ostatní (kniha)	610 856
geo	geo	Geofyzikální data	102 400
news	news	USENET batch soubor	377 109
obj1	obj1	Objektový kód pro VAX	21 504
obj2	obj2	Objektový kód pro Mac	246 814
paper1	paper1	Technická dokumentace	53 161
paper2	paper2	Technická dokumentace	82 199
paper3	paper3	Technická dokumentace	46 526
paper4	paper4	Technická dokumentace	13 286
paper5	paper5	Technická dokumentace	11 954
paper6	paper6	Technická dokumentace	38 105
pic	pic	Černobílý obrázek (fax)	513 216
progc	progc	C - zdrojový kód	39 611
progl	progl	LISP – zdrojový kód	71 646
progp	progp	Pascal – zdrojový kód	49 379
trans	trans	Transcript	93 695

Tabulka 15: Silesia Corpus

Název souboru	Popis	Velikost [Byte]
dickens	Anglický text - Charles Dickens	10 192 446
mozilla	Spustitelný soubor - Mozilla 1.0	51 220 480
mr	Snímek magnetické rezonance	9 970 564
nci	Chemická databáze struktůr	33 553 445
ooffice	Open Office.org DLL 1.01	6 152 192
osdb	Vzorek MySQL databáze	10 085 684
reymont	Polský text (pdf) - Władysław Reymont	6 627 202
samba	Zdrojový kód - Samba 2-2.3	21 606 400
sao	Katalog hvězd SAO - binární data	7 251 944
webster	Webstrův slovník - html	41 458 703
xml	Sbírka XML souborů - html	5 345 208
x-ray	Sníměk rentgenu	8 474 240

C Výsledky testování

Tabulka 16: Výsledky testování - čas výpočtů - první část

Název souboru	Kompresní proces [ms]					Dekompresní proces [ms]				
	GRP	MTF	RLE	SPL	Σ	GRP	MTF	RLE	SPL	Σ
alice29.txt	1 168	3	5	35	1 211	1 136	2	<1	23	1 161
asyoulik.txt	1 069	3	2	26	1 100	901	2	<1	24	927
cp.html	40	1	<1	5	46	62	<1	<1	3	65
fields.c	16	<1	<1	2	18	19	<1	<1	1	20
grammar.lsp	2	<1	<1	<1	2	4	<1	<1	<1	4
kennedy.xls	1 284	26	10	121	1 441	1 264	19	2	81	1 366
lcet10.txt	1 197	10	10	96	1 314	1 163	10	5	64	1 241
plrabn12.txt	1 172	11	9	96	1 287	1 192	8	8	66	1 274
ptt5	1 169	3	5	28	1 205	1 191	2	1	21	1 216
sum	96	2	<1	7	105	138	1	<1	5	144
xargs.1	5	<1	<1	<1	5	4	<1	<1	<1	4
a.txt	2	<1	2	<1	5	<1	<1	<1	<1	1
aaa.txt	639	<1	1	<1	639	711	<1	<1	<1	711
alphabet.txt	568	<1	1	<1	569	670	<1	<1	<1	670
random.txt	553	5	2	25	586	676	2	<1	23	700
bible.txt	3 964	84	87	694	4 829	3 988	52	16	561	4 618
E.coli	4 324	43	121	375	4 864	4 520	40	34	321	4 916
world192.txt	2 624	70	52	470	3 216	2 434	43	11	391	2 878
pi.txt	1 300	15	18	156	1 489	1 290	16	8	111	1 424
bib	906	3	3	22	936	532	2	<1	21	555
book1	1 213	18	18	157	1 406	1 211	13	4	128	1 356
book2	1 184	16	14	141	1 356	1 195	13	5	112	1 324
geo	557	6	<1	17	580	754	3	<1	16	773
news	1 152	13	6	94	1 265	1 154	8	13	59	1 234
obj1	48	1	<1	4	53	52	1	<1	3	56
obj2	1 112	15	1	66	1 194	1 132	8	<1	48	1 188
paper1	208	1	1	10	220	195	1	<1	8	204
paper2	588	2	1	15	606	521	1	1	11	534
paper3	209	1	<1	8	218	136	<1	<1	8	145
paper4	20	<1	<1	2	22	18	<1	<1	1	19
paper5	16	<1	<1	2	18	16	<1	<1	1	17
paper6	129	1	<1	7	137	93	<1	<1	6	99
pic	1 172	3	5	28	1 208	1 189	2	1	22	1 214
progc	173	1	<1	8	182	116	<1	<1	7	124
progl	330	1	1	17	350	418	1	<1	11	429
progp	168	1	<1	11	180	139	2	<1	10	151
trans	491	3	1	24	519	626	2	<1	21	649

Tabulka 17: Výsledky testování - čas výpočtů - druhá část

Název souboru	Kompresní proces [s]					Dekompresní proces [s]				
	GRP	MTF	RLE	SPL	Σ	GRP	MTF	RLE	SPL	Σ
dickens	10,07	0,12	0,18	1,80	12,17	9,94	0,09	0,04	1,45	11,52
mozilla	47,67	0,78	0,38	8,96	57,78	47,89	0,41	0,02	7,20	55,51
mr	9,98	0,19	0,08	1,16	11,41	9,92	0,11	<0,01	0,90	10,93
nci	31,05	0,05	0,35	2,55	34,00	31,08	0,04	0,14	2,04	33,30
ooffice	5,84	0,36	0,05	1,42	7,67	5,93	0,21	<0,01	1,21	7,34
osdb	9,88	0,34	0,08	2,42	12,72	9,96	0,19	<0,01	2,03	12,19
reymont	6,30	0,10	0,06	1,02	7,47	6,26	0,08	<0,01	0,78	7,13
samba	19,99	0,30	0,16	3,77	24,21	19,96	0,17	<0,01	2,95	23,09
sao	6,79	0,45	0,08	1,70	9,02	6,72	0,28	<0,01	1,40	8,40
webster	38,70	0,17	0,55	7,30	46,72	39,04	0,09	0,17	5,77	45,07
x-ray	7,96	0,44	0,07	1,76	10,23	7,97	0,25	<0,01	1,46	9,68
xml	5,67	0,09	0,11	0,81	6,68	5,58	0,08	0,02	0,66	6,34

Uvedené hodnoty v tabulkách 16 a 17 jsou reprezentovány průměrem v 10 měřeních.
Tabulka 16 uvádí hodnoty v milisekundách. Tabulka 17 uvádí hodnoty v sekundách.
Hodnota <1 označuje čas výpočtů menší než 1 ms.
Hodnota <0,01 označuje čas výpočtů menší než 10 ms.

Tabulka 18: Výsledky testování - komprese - první část

Název souboru	α	f	s	bps^*
alice29.txt	0,670	1,492	0,330	5,363
asyoulik.txt	0,700	1,428	0,300	5,602
cp.html	0,785	1,274	0,215	6,282
fields.c	0,769	1,301	0,231	6,150
grammar.lsp	0,746	1,341	0,254	5,966
kennedy.xls	0,333	3,000	0,667	2,666
lcet10.txt	0,685	1,461	0,315	5,477
plrabn12.txt	0,662	1,510	0,338	5,297
ptt5	0,177	5,666	0,823	1,412
sum	0,771	1,298	0,229	6,165
xargs.1	0,793	1,262	0,207	6,340
a.txt	24,000	0,042	-23,000	192,000
aaa.txt	0,004	266,667	0,996	0,030
alphabet.txt	0,051	19,573	0,949	0,409
random.txt	0,902	1,109	0,098	7,212
bible.txt	0,622	1,607	0,378	4,978
E.coli	0,320	3,124	0,680	2,561
world192.txt	0,712	1,404	0,288	5,699
pi.txt	0,498	2,008	0,502	3,985

Tabulka 19: Výsledky testování - komprese - druhá část

Název souboru	α	f	s	bps^*
bib	0,705	1,418	0,295	5,641
book1	0,687	1,456	0,313	5,494
book2	0,728	1,373	0,272	5,826
geo	0,712	1,404	0,288	5,699
news	0,802	1,247	0,198	6,417
obj1	0,919	1,089	0,081	7,349
obj2	0,870	1,150	0,130	6,958
paper1	0,776	1,288	0,224	6,212
paper2	0,706	1,416	0,294	5,650
paper3	0,734	1,363	0,266	5,869
paper4	0,760	1,316	0,240	6,077
paper5	0,800	1,249	0,200	6,403
paper6	0,785	1,275	0,215	6,277
pic	0,177	5,666	0,823	1,412
progc	0,797	1,254	0,203	6,377
progl	0,682	1,466	0,318	5,455
progp	0,715	1,400	0,285	5,716
trans	0,767	1,303	0,233	6,138
dickens	0,667	1,499	0,333	5,337
mozilla	0,685	1,460	0,315	5,481
mr	0,431	2,322	0,569	3,445
nci	0,284	3,525	0,716	2,270
ooffice	0,923	1,084	0,077	7,383
osdb	0,959	1,043	0,041	7,669
reymont	0,557	1,794	0,443	4,459
samba	0,674	1,484	0,326	5,393
sao	0,959	1,043	0,041	7,671
webster	0,673	1,486	0,327	5,383
x-ray	0,828	1,208	0,172	6,625
xml	0,569	1,757	0,431	4,553

* bps - Označuje bit per symbol, bit per Byte. Tedy počet potřebných bitů k reprezentaci jednoho symbolu nebo Bytu.

Tabulka 20: Výsledky testování transponovaných experimentálních variant

Název souboru	Varianty [f]				
	V1	EV5	EV6	ST1	ST2
alice29.txt	1,492	1,738	1,673	1,524	1,740
asyoulik.txt	1,428	1,528	1,594	1,487	1,673

Tabulka 21: Výsledky testování transponované experimentální varianty - ST2

Název souboru	*f*	*bps*
alice29.txt	1,74	4,59
asyoulik.txt	1,67	4,77
cp.html	1,48	5,40
fields.c	1,62	4,93
grammar.lsp	1,69	4,74
lcet10.txt	1,81	4,66
plrabn12.txt	1,81	4,43
ptt5	5,73	1,40
sum	1,70	4,68
xargs.1	1,59	5,02

Tabulka 22: Squash benchmark - soubor *alice29.txt*

#	Kompresní metoda	Verze	Úroveň	*f*
1	zpaq	zpaq	5	4, 07
2	zpaq	zpaq	4	3, 91
⋮	⋮	⋮	⋮	⋮
241	lzo	lzo1x	1	1, 75
242	**grp**	**grp-st2**	**-**	1, 74
243	lzo	lzo1f	2	1, 73
242	pithy	pithy	2	1, 73
⋮	⋮	⋮	⋮	⋮
278	lz4	lz4	1	1, 07

D Příloha na CD/DVD

- **src** - zdrojové kódy implementace
  - **source-console** - zdrojové kódy konzolové aplikace
  - **source-library** - zdrojové kódy kompresní knihovny
  - **vs-project** - vytvořený Visual Studio projekt
- **reh0074-dp.pdf** - text této diplomové práce